

Toshiba Global Commerce Solutions 4690 OS



# Programming Guide

*Version 6 Release 4*



Toshiba Global Commerce Solutions 4690 OS



# Programming Guide

*Version 6 Release 4*

**Note**

Before using this information and the product it supports, be sure to read Safety Information- Read This First, Warranty Information, Uninterruptible Power Supply Information and the information under Appendix D, "Notices," on page 751.

**September 2013**

This edition applies to Version 6 Release 4 of the licensed program Toshiba 4690 OS (program number 5639-P70) and to all subsequent releases and modifications until otherwise indicated in new editions.

If you send information to Toshiba Global Commerce Solutions (Toshiba), you grant Toshiba nonexclusive right to use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

© Copyright Toshiba Global Commerce Solutions, Inc. 2013.

© Copyright IBM Corporation 2010, 2012

---

# Contents

<b>Figures</b> . . . . .	xxi
<b>Tables</b> . . . . .	xxiii
<b>Safety</b> . . . . .	xxv
<b>About this guide</b> . . . . .	xxvii
Who should read this guide . . . . .	xxvii
Terminal models. . . . .	xxvii
Where to find more information . . . . .	xxviii
4690 V6 library . . . . .	xxviii
Notice statements . . . . .	xxviii

---

## Part 1. The operating system environment . . . . . 1

<b>Chapter 1. The operating system environment.</b> . . . . .	3
Operating modes . . . . .	3
Classic Mode . . . . .	3
Enhanced Mode . . . . .	3
System capabilities. . . . .	4
Concurrent operations. . . . .	4
Operator interface . . . . .	4
Changing the signon screen . . . . .	4
Programming auto sign off . . . . .	5
Wait for system console keyboard and mouse inactivity . . . . .	5
Wait for system console keyboard and mouse activity . . . . .	5
Sign off the active user . . . . .	5
Disconnect the active user . . . . .	6
Example. . . . .	6
Programming auto sign off for system and auxiliary consoles . . . . .	6
Start inactivity monitoring . . . . .	6
Example. . . . .	7
Stop inactivity monitoring . . . . .	7
Signoff an active user . . . . .	7
Disconnect an active user . . . . .	7
Communications. . . . .	7
Files . . . . .	8
Problem-determination aids. . . . .	8
Utilities . . . . .	8
Additional products . . . . .	9
Store controller backup . . . . .	9
<b>Chapter 2. Managing files</b> . . . . .	11
Selecting file types . . . . .	11
Random . . . . .	11
Direct . . . . .	12
Keyed . . . . .	12
Sequential . . . . .	14
Naming files and subdirectories. . . . .	16
Rules for naming subdirectories and files on FAT systems . . . . .	17
Rules for naming subdirectories and files on VFS drives. . . . .	20
Using logical names . . . . .	20
Defining logical file names. . . . .	21

Logical file names on a LAN (MCF Network) system . . . . .	22
Logical names table . . . . .	22
Using logical names to specify destination service access points . . . . .	23
Accessing distributed files in store controller and terminal applications . . . . .	24
Step 1. Define the file . . . . .	24
Step 2. Create the file . . . . .	25
Step 3. Access the file from the user program . . . . .	26
Using file names in store controller applications . . . . .	26
Using file names in terminal applications . . . . .	26
Using node names to access files . . . . .	27
Logging distribution errors . . . . .	28
Using long file names on VFS systems . . . . .	28
VFS translation . . . . .	28
Name server database . . . . .	28
Restrictions when accessing long file name support . . . . .	28
Performing file functions . . . . .	29
Creating files . . . . .	29
Deleting files . . . . .	30
Accessing files . . . . .	30
Ending access to files . . . . .	31
Sharing files . . . . .	31
Copying files . . . . .	33
Renaming files . . . . .	33
Protecting files . . . . .	33
Protecting subdirectories . . . . .	34
Enabling and disabling file and subdirectory security . . . . .	35
Reading a file record . . . . .	35
Writing a file record . . . . .	35
Ensuring data integrity across power line disturbances . . . . .	37
Design considerations for file performance . . . . .	37
Disabling the write verify function for your hard disk drive . . . . .	38
Keyed files . . . . .	38
<b>Chapter 3. Programming Terminal I/O Devices . . . . .</b>	<b>39</b>
2x20 Displays . . . . .	39
Characteristics . . . . .	39
Functions Your Application Performs . . . . .	40
Accessing the Display . . . . .	40
Clearing the Display . . . . .	40
Writing Characters to the Display . . . . .	40
Determining Display information . . . . .	42
Reading from the Display . . . . .	42
Programming Hints for 2x20 Displays . . . . .	43
About DBCS Enabling . . . . .	43
Example . . . . .	43
Shopper Display . . . . .	44
Characteristics . . . . .	44
Functions Your Application Performs . . . . .	45
Accessing the Shopper Display . . . . .	45
Clearing the Shopper Display . . . . .	45
Writing Characters to the Shopper Display . . . . .	45
Setting the Guidance Lights on the Display . . . . .	45
Reading from the Display . . . . .	45
Determining Shopper Display Status . . . . .	45
Example . . . . .	45
Video Display . . . . .	47

Characteristics . . . . .	47
Functions Your Application Performs . . . . .	49
Accessing the Video Display . . . . .	50
Clearing the Video Display . . . . .	50
Changing the Video Display Format . . . . .	50
Writing to the Video Display . . . . .	51
Determining Video Display Information . . . . .	52
Reading from the Video Display . . . . .	52
Restrictions Using Command Stacking . . . . .	58
Example Program Using Command Stacking . . . . .	58
Example Program Source Code . . . . .	59
Cash Drawer Driver . . . . .	60
Characteristics . . . . .	60
Functions your Application Performs . . . . .	61
Accessing the Cash Drawer or Alarm. . . . .	61
Controlling a Cash Drawer or Alarm . . . . .	61
Obtaining Cash Drawer or Alarm Status. . . . .	61
Example . . . . .	62
Coin Dispenser Driver . . . . .	62
Characteristics . . . . .	62
Accessing the Coin Dispenser . . . . .	63
Dispensing Coins . . . . .	63
Example . . . . .	63
I/O Processor . . . . .	64
Characteristics . . . . .	64
Input Devices on Your System . . . . .	65
I/O Processor Functions . . . . .	65
Input Sequence Tables . . . . .	65
Functions Your Application Performs . . . . .	72
Additional I/O Processor Features . . . . .	80
Magnetic Stripe Reader Driver (MSR) . . . . .	84
Characteristics of the Single-Track Magnetic Stripe Reader . . . . .	85
Characteristics of the Dual-Track Magnetic Stripe Reader . . . . .	85
Characteristics of the Three-Track Magnetic Stripe Reader. . . . .	85
Characteristics of the JIS-II Magnetic Stripe Reader . . . . .	85
Data Formats and Error Reporting. . . . .	86
Restrictions of Single- and Multi-Track MSRs. . . . .	88
Functions Your Application Performs . . . . .	88
Accessing the MSR . . . . .	89
Preparing to Receive Data from the MSR . . . . .	89
Waiting for Received Data. . . . .	89
Receiving Data . . . . .	89
Data Characteristics Common to all MSRs. . . . .	89
Disallowing MSR Data . . . . .	90
Determining Status of the MSR . . . . .	90
Single-Track MSR Example . . . . .	91
Dual-Track MSR Example . . . . .	92
Three-Track MSR Example . . . . .	93
JUCC MSR Example. . . . .	95
Printer Driver Model 2 . . . . .	99
Characteristics . . . . .	99
Functions Your Application Performs . . . . .	100
Accessing the Printer . . . . .	100
Preparing a Line To Print. . . . .	100
Printing a Line of Text at the Printer. . . . .	100
Controlling the Document Insert Station . . . . .	100

Determining the Printer Status . . . . .	101
Printing Checks . . . . .	101
Logo Printing . . . . .	103
Printer Driver Model 3 or 4/4A . . . . .	106
Characteristics . . . . .	106
Compatibility with Applications Written for the Model 1 and 2 Printers . . . . .	106
Functions Your Application Performs . . . . .	107
Accessing the Printer . . . . .	107
Preparing a Line to Print . . . . .	107
Printing a Line of Text at the Printer. . . . .	107
Controlling the Document Insert Station . . . . .	109
Determining the Printer Status . . . . .	113
Preventing Unnecessary Reprints . . . . .	113
Receipt Paper Cutter . . . . .	114
Printer Home Sensors . . . . .	114
Left Home Command . . . . .	114
Printing Checks . . . . .	114
Logo Printing . . . . .	114
Determining When Printing is Complete . . . . .	115
Performance Considerations . . . . .	115
Read Commands . . . . .	115
Magnetic Ink Character Recognition Support for Printers Model 3 and 4/4A . . . . .	118
MICR Data Format . . . . .	118
Using the MICR Reader . . . . .	119
Determining If a MICR Is Installed . . . . .	119
Reading from the MICR . . . . .	119
Understanding MICR Parsing Routines . . . . .	120
4610 Printer . . . . .	120
Characteristics . . . . .	120
Compatibility with Applications Written for the Model 3 and 4/4A Printers . . . . .	121
Writing a Common Application for Different Printers . . . . .	122
Functions Your Application Performs . . . . .	122
Accessing the Printer . . . . .	122
Waiting for Printer . . . . .	122
Determining if Data Has Been Received . . . . .	122
Reading Data From the Printer . . . . .	122
Recovering From an Error . . . . .	123
Preparing a Line To Print. . . . .	123
Printing a Line of Text at the Printer. . . . .	123
Controlling the Document Insert Station . . . . .	124
Determining the Printer Status. . . . .	124
Receipt Paper Cutter . . . . .	125
Printer Home Sensors. . . . .	125
Printing Checks . . . . .	125
Logo Printing . . . . .	126
Determining When Printing is Complete . . . . .	127
4689 Printer . . . . .	127
Characteristics . . . . .	127
Compatibility with Applications Written for 4610 Printers . . . . .	128
Accessing the Printer . . . . .	128
Waiting for the Printer . . . . .	128
Determining if Data Has Been Received . . . . .	129
Recovering From an Error . . . . .	129
Writing to the Printer . . . . .	129
Returning Invalid Data. . . . .	129
Determining the Printer Status. . . . .	129



Determining When Printing is Complete . . . . .	129
Line Count . . . . .	130
4689 Native Mode Support . . . . .	130
Switching to 4689 Native Mode . . . . .	130
Determining the Printer Mode . . . . .	130
Commands . . . . .	130
Fiscal Printer Support . . . . .	137
Application Programming for the Operating System Fiscal System . . . . .	138
Reading from the Model 3 Fiscal Printer . . . . .	138
Using the FISREAD call . . . . .	139
Scale Driver . . . . .	139
Characteristics . . . . .	139
Accessing the Scale . . . . .	139
Receiving Data . . . . .	139
Example . . . . .	139
Serial I/O Communications Driver . . . . .	141
Characteristics . . . . .	141
Functions Your Application Performs . . . . .	141
Accessing the Serial I/O Port . . . . .	141
Overview of Receiving Data. . . . .	142
Preparing to Receive Data from the Device . . . . .	142
Waiting for Received Data . . . . .	143
Receiving Data from the Device . . . . .	143
Determining Serial I/O Port Status . . . . .	143
Preparing to Transmit Data to the Device. . . . .	144
Transmitting Data to the Device . . . . .	144
Sending a Break to the Device . . . . .	144
Simultaneous Receive and Transmit . . . . .	144
Example . . . . .	144
Tone Driver. . . . .	146
Characteristics . . . . .	146
Functions Your Application Performs . . . . .	146
Accessing the Tone. . . . .	146
Generating a Tone . . . . .	146
Example . . . . .	147
Totals Retention Driver . . . . .	147
Characteristics . . . . .	147
Accessing the Totals Retention Driver . . . . .	148
Accessing Totals Retention in Direct Mode . . . . .	148
Reading Data in Direct Mode . . . . .	148
Writing Data in Direct Mode. . . . .	148
Accessing Totals Retention in Sequential Mode . . . . .	148
Specifying the Address in Sequential Mode . . . . .	149
Reading Data in Sequential Mode . . . . .	149
Writing Data in Sequential Mode . . . . .	149
Example . . . . .	149
<b>Chapter 4. Using error recovery procedures and facilities . . . . .</b>	<b>153</b>
Error recovery options. . . . .	153
Error functions and statements . . . . .	153
ON ERROR statement . . . . .	153
ON ASYNC ERROR CALL statement . . . . .	154
IF END# statement . . . . .	154
RESUME statement . . . . .	154
ERR function . . . . .	155
ERRL function. . . . .	155

ERRF% function . . . . .	155
ERRN function . . . . .	155
Logging system errors. . . . .	155
System log . . . . .	155
System messages . . . . .	156
Event log access. . . . .	157
Record formats . . . . .	158
Audible alarm . . . . .	159
Setting the audible alarm function . . . . .	159
Activating the audible alarm function . . . . .	160
Using the audible alarm . . . . .	160
Deactivating the audible alarm function . . . . .	161
Distribution exception log. . . . .	161
Power line disturbance recovery . . . . .	161
Store controller PLD recovery . . . . .	161
Terminal PLD Recovery . . . . .	161
Storage retention . . . . .	162
Terminal power on/off . . . . .	162
Terminal power on/off for the Mod1 terminal with storage retention enabled . . . . .	162
Terminal power on/off for the Mod2 terminal. . . . .	162
Disk and file error recovery . . . . .	162
Error recovery for I/O devices . . . . .	165
Application responses . . . . .	165
 <b>Chapter 5. User Application Considerations with a LAN (MCF Network)</b>	167
Using TCLOSE to Close Application Data Files . . . . .	167
Application Read Restrictions . . . . .	167
Spool Files on a LAN (MCF Network) System . . . . .	167
Erasing the Spool File . . . . .	168
Forcing the Spool File to be Erased. . . . .	169
Using Drive D: for the Spool File . . . . .	169
Saving Data That Cannot be Successfully Despooled . . . . .	169
Effects of Activating the Alternate File Server on Despooling. . . . .	169
Reactivating the Configured File Server . . . . .	170
Record Sizes . . . . .	170
Using the Application Program Interface (OS/2) . . . . .	170
Example of a C/2 C-Language Program . . . . .	171
Using the Application Program Interface (DOS) . . . . .	171
Example of a DOS C-Language Program. . . . .	172
Example of a 4680 BASIC Program. . . . .	173

---

## Part 2. Utilities . . . . . 175

<b>Chapter 6. Using the Keyed File Utility</b> . . . . .	177
Accessing the Keyed File Utility . . . . .	177
Using the Keyed File Utility from the Host . . . . .	177
Creating a Keyed File . . . . .	178
Creating a Direct File . . . . .	178
Reporting Keyed File Statistics . . . . .	179
Creating an Empty Keyed File . . . . .	179
Chaining Statistics from a Direct File . . . . .	179
Rebuilding a Keyed File . . . . .	179
Using the Keyed File Utility in a Batch File . . . . .	179
Creating a Keyed File from a Direct File . . . . .	179
Creating a Direct File from a Keyed File . . . . .	180
Creating an Empty File (Batch Method) . . . . .	181

Reporting Keyed File Statistics . . . . .	182
Reporting Chaining Statistics from a Direct File . . . . .	182
Restarting from a Checkpoint . . . . .	183
Canceling (Erasing) a Checkpoint . . . . .	183
Keyed File Utility Working Files . . . . .	183
Hashing Algorithms . . . . .	183
Specifying Algorithms for Files . . . . .	184
Verifying Definitions. . . . .	185
Internal Processes of Keyed Files . . . . .	185
Folding Algorithm . . . . .	185
The XOR Rotation Hashing Algorithm . . . . .	187
Example of the XOR Rotation Hashing Algorithm . . . . .	187
Polynomial Hashing Algorithm . . . . .	189
Example C-language Program . . . . .	191
Record Chaining . . . . .	191
Keyed File Control Record . . . . .	194
 <b>Chapter 7. Using the Input Sequence Table Build Utility . . . . .</b>	<b>197</b>
Input Sequence Tables . . . . .	197
Using the Input State Table Utility . . . . .	197
Tables Maintained by the Utility . . . . .	197
Running the Input Sequence Table Utility . . . . .	197
Input Sequence Table Utility on a LAN (MCF Network) System . . . . .	197
Input Sequence Table Utility Options . . . . .	198
Table Naming Conventions . . . . .	199
Notes on Using the Utility . . . . .	199
 <b>Chapter 8. Using the LIB86 Library Utility . . . . .</b>	<b>201</b>
Using LIB86 Command-Line Options . . . . .	201
Creating a Library File . . . . .	202
Appending an Existing Library . . . . .	203
Replacing Library Modules . . . . .	203
Deleting Library Modules . . . . .	203
Selecting Modules . . . . .	204
Displaying Library Information . . . . .	204
Accessing Files in Other Directories. . . . .	205
 <b>Chapter 9. Using the Linker Utility and the POSTLINK Utility . . . . .</b>	<b>207</b>
The Linker Utility . . . . .	207
LINK86 Command Syntax . . . . .	208
Linking With Shared Run-time Libraries . . . . .	209
LINK86 Command Options . . . . .	209
Use of Link Path Variables to Search Other Directories . . . . .	215
How Various Search Priorities Relate . . . . .	216
Use of ERRORLEVEL Test . . . . .	216
Overlays . . . . .	216
Overlay Command Line Syntax . . . . .	217
The POSTLINK Utility . . . . .	218
Invoking the POSTLINK Utility . . . . .	218
Use of ERRORLEVEL Test . . . . .	219
 <b>Chapter 10. Using the Print Spooler Utility . . . . .</b>	<b>221</b>
Obtaining Job Status after a TCLOSE . . . . .	221
Issuing a Command to the Print Spooler . . . . .	222
Using Special Commands . . . . .	223
Error Return Codes for the Print Spooler . . . . .	225

<b>Chapter 11. Using the Disk Surface Analysis Utility</b>	227
Introduction to the Disk Surface Analysis Utility	227
Disk Surface Analysis Utility	227
Parameter Descriptions for ADXCSW0L	228
Command Formats for ADXCSW0L	228
Example 1	228
Example 2	229
Example 3	229
Example 4	230
Using the Disk Surface Analysis Utility to Recover Data	231
Using the Time Frame Indicators	232
Case Examples of Disk Recovery	232
Non-LAN (MCF Network), Defect in .286 File	232
LAN (MCF Network), Defect in .286 File on Master Controller	233
LAN (MCF Network), Defect in .286 File on Alternate Master Controller	233
Non-LAN (MCF Network), Defect in Unallocated Space	234
LAN (MCF Network), Defect in Unallocated Space on Master Controller	234
LAN (MCF Network), Defect in Unallocated Space on Alternate Master Controller	235
Non-LAN (MCF Network), Defect in Item Record File	235
LAN (MCF Network), Defect in Item Record File on Master Controller	236
LAN (MCF Network), Defect in Item Record File on Alternate Master	236
Non-LAN (MCF Network), Defect Near End of TLOG	237
LAN (MCF Network), Defect Near End of TLOG on File Server	237
LAN (MCF Network), Defect Near End of TLOG on Alternate File Server	237
<b>Chapter 12. Using the IPL Command Processor</b>	239
Example of a Command File	240
<b>Chapter 13. Using the Staged IPL Utility</b>	241
Requirements	241
Capabilities	241
TCC Network Considerations	241
Applications Only	241
Incompatible Software Levels	241
Application Interface	242
Apply Software Maintenance	242
RCP Command	242
Error Recovery	243
Recommended Use	243
Loading Terminal Storage	246
Enable Terminal IPL	246
Disable Terminal IPL	246
Messages	246
<b>Chapter 14. Using the Java Terminal Offline Function</b>	249
Understanding the Java Terminal Offline Function	249
Requirements for Applications Using the JavaTOF Solution	249
Resource Creation	250
Dependency Checking	250
Dependency Checking Limitations	250
Using the JavaTOF Classes	250
Setting up the Java Classpath for Running the JavaTOF Solution	251
JavaTOF Property File	251
Resource Creation	252
Dependency Checking	253

Application Properties . . . . .	255
The JavaTOF Property File Editor . . . . .	256
Procedure Overview . . . . .	258
Creating the Resource File . . . . .	259
Creating the Dependency Checking Output File . . . . .	259
Running the Application . . . . .	260
Font Preloading . . . . .	260
Using JavaTOF with Java 2 (Single Application) . . . . .	261
Boot Archives . . . . .	261
Procedure for Using JavaTOF with Java 2 . . . . .	262
Using JavaTOF with Multi-App. . . . .	263
Java 2 Terminal Classpath Setup. . . . .	263
JavaTOF Properties Files . . . . .	264
Multi-App Properties File . . . . .	264
Java 2 Class and Parameters . . . . .	265
Terminal RAM Disk Preload (Java 2) . . . . .	265
Creating a BAT File. . . . .	266
JavaTOF Procedural Enhancements . . . . .	266
 <b>Chapter 15. Using the Multiple File Archiver Utility . . . . .</b>	 269
Compressing, Combining, and Archiving Files . . . . .	269
Mapping the Combine File . . . . .	270
Splitting Files Out of a Combine File . . . . .	271
Splitting a Given List of Files from a Combine File . . . . .	272
Log Files . . . . .	272
Return Codes . . . . .	272

---

## Part 3. Applications (designing and using) . . . . . 275

<b>Chapter 16. Designing applications with 4680 BASIC . . . . .</b>	<b>277</b>
Types of applications . . . . .	277
Interactive applications . . . . .	277
Background applications . . . . .	277
Application size . . . . .	278
Memory models . . . . .	278
Code size . . . . .	278
Data size . . . . .	279
File size . . . . .	280
Application priorities . . . . .	280
File access priorities in terminal applications . . . . .	280
Starting a background application . . . . .	280
ADXSTART. . . . .	280
ADXPSTAR . . . . .	281
System authorization . . . . .	282
ADXAUTH . . . . .	283
ADXCRIPT . . . . .	285
ADXDATE (retrieving a 4-digit year system date) . . . . .	286
Communicating between applications . . . . .	286
Using pipes. . . . .	286
Using the 4690 Java-BASIC API . . . . .	291
Using Application Services . . . . .	293
Store controller and terminal application services . . . . .	293
Chaining applications . . . . .	313
Chaining to overlays . . . . .	313
Chaining to directly executable modules . . . . .	314
RAM disk files. . . . .	314

Considerations for installing RAM disks . . . . .	314
Accessing RAM disk files from controller applications . . . . .	315
Accessing controller RAM disk files from terminal applications . . . . .	315
Accessing terminal RAM disk files from terminal applications . . . . .	315
Store controller application services . . . . .	317
ADXFILES (canceling the shared use of a file). . . . .	317
ADXCSEOL (store closed query application). . . . .	320
ADXEXIT (set the ERRORLEVEL VALUE) . . . . .	321
ADXSERCL (closing an application service). . . . .	321
Terminal application services . . . . .	322
ADXDIR (listing terminal RAM disk files) . . . . .	322
Extended memory management for the Point of Sale terminal . . . . .	324
Terminal hard disk files . . . . .	332
Considerations for installing a hard disk in the terminal. . . . .	332
Accessing hard disk files from terminal applications . . . . .	332
<b>Chapter 17. Designing Applications with Other Languages . . . . .</b>	<b>333</b>
Operating System Interfaces for C and COBOL . . . . .	333
Using the 16-Bit C Language Interface. . . . .	334
Using the 32-Bit C Language Interface. . . . .	334
Using the COBOL Language Interfaces . . . . .	334
Disk File Management. . . . .	335
File Access . . . . .	335
Pipe Management . . . . .	336
File and Pipe Services . . . . .	338
Pipe Routing Services. . . . .	361
Communications Services . . . . .	364
Specialized Services . . . . .	364
Miscellaneous Services . . . . .	384
Guidelines and Restrictions for 16-Bit Assembly Language Applications . . . . .	389
<b>Chapter 18. Using IBM DOS Applications . . . . .</b>	<b>391</b>
Starting Applications in the DOS Environment . . . . .	391
DOS Program Memory Allocation. . . . .	392
Allocating Memory Using ADDMEM. . . . .	392
Optional Emulator Reports . . . . .	392
Selecting Reports Using EOPTIONS . . . . .	393
Report "a"—Errors . . . . .	393
Report "b"—Startup Data. . . . .	393
Report "c"—Interrupt Trace . . . . .	393
Report "d"—OUT, OUTS, IN, INS Trace . . . . .	393
DOS BIOS Calls and Software Interrupts . . . . .	394
BIOS Calls . . . . .	394
Software Interrupts . . . . .	395
DOS Function Calls. . . . .	395
Emulator Messages. . . . .	396
<b>Chapter 19. Designing Terminal Applications With C Language . . . . .</b>	<b>397</b>
Operating System Interfaces for C . . . . .	397
16-Bit CPI Libraries. . . . .	397
32-Bit CPI Libraries. . . . .	397
Error Handling. . . . .	398
Asynchronous Error Interrupt Handling. . . . .	398
Using the Heap Manager. . . . .	398
16-Bit Restrictions . . . . .	398
Header File. . . . .	399

Compiling and Linking . . . . .	399
File Services . . . . .	399
File Types . . . . .	399
Access Modes . . . . .	400
File Security . . . . .	400
File Pointers . . . . .	400
Common File Functions . . . . .	401
File and Pipe Services . . . . .	403
Pipe Routing Services . . . . .	412
Create a Pipe Routing Services Pipe . . . . .	412
Waiting for Data in a Pipe Routing Services Pipe . . . . .	413
Reading a Pipe Routing Services Pipe . . . . .	413
Close a Pipe Routing Services Pipe . . . . .	413
Open the Pipe Routing Services Driver . . . . .	413
Writing to a Pipe Routing Services Pipe . . . . .	414
Conditional Write to a Pipe Routing Services Pipe . . . . .	414
Close the Pipe Routing Services Driver . . . . .	415
CPI Device Interface . . . . .	415
Open a Device . . . . .	415
Open the I/O Processor . . . . .	416
Open a Serial I/O Session . . . . .	417
Close a Device . . . . .	418
Position the Cursor . . . . .	418
Clear the Display . . . . .	419
Get Device Status Bytes . . . . .	419
Set Device Status Bytes . . . . .	419
Load State Tables . . . . .	420
Lock a Device . . . . .	420
Unlock a Device . . . . .	420
Define an Asynchronous Error Handler . . . . .	421
Set Totals Retention Offset . . . . .	423
Get Totals Retention Offset . . . . .	423
Read from a Device . . . . .	423
Read Direct from Totals Retention . . . . .	425
Wait for Event Completion . . . . .	425
Write to a Device . . . . .	425
Write Direct to Totals Retention . . . . .	427
Write Bitmap Data to the Printer . . . . .	427
Flush Pending Writes to the Printer . . . . .	427
Read Data from the Fiscal Printer . . . . .	428
Miscellaneous Routines . . . . .	428
Allocate Heap Space . . . . .	428
Free Heap Space . . . . .	429
Query Available Heap Space . . . . .	429
Query Contiguous Available Heap Space . . . . .	429
Chain to Another Program . . . . .	429
Receive Parameters From Chaining Program . . . . .	430
Get System Date . . . . .	431
Get System Time . . . . .	431
Disable Software Interrupts . . . . .	431
Enable Software Interrupts . . . . .	431
Pack Data . . . . .	431
Unpack Data . . . . .	432
Requesting Operating System Memory . . . . .	432
Freeing Storage . . . . .	434
Suspending Processing for Indicated Duration . . . . .	434



Waiting for Data . . . . .	434
Converting Hex to ASCII . . . . .	435
Terminal Application Services . . . . .	436
ADX_TSERVICE (Requesting an Application Service) . . . . .	436
Switching between the terminal Java, terminal application, and enhanced mode graphical extensions screens . . . . .	441
ADX_TERROR (Log an Application Event) . . . . .	442
ADX_TDIR (Listing Terminal RAM/Hard Disk Files) . . . . .	443
ADX_TCOPYF (Copying Disk Files). . . . .	444
ADX_TCRYPT (Encrypting a Password) . . . . .	444
CPI Extended Memory Management for the 469x POS Terminal . . . . .	445
Error Handling. . . . .	445
Data Buffers . . . . .	445
Allocating a Memory File . . . . .	445
Accessing a Shared Memory File. . . . .	446
Gaining Mutually Exclusive Access to Shared Memory . . . . .	446
Releasing Exclusive Access to Shared Memory . . . . .	446
Freeing a Memory File . . . . .	446
Querying Available Free Memory. . . . .	447
Writing to a Memory File . . . . .	447
Reading From a Memory File . . . . .	447
Binary Search for Matching Field. . . . .	448
Sequential Search for Matching Field . . . . .	448
Writing Data to a Memory File in Ascending Order by Key . . . . .	448
Deleting Data from a Memory File Sequenced by Key . . . . .	449
Clearing a Memory File . . . . .	449
Error Codes . . . . .	449
Product Signature . . . . .	450
Validation of Application Buffers . . . . .	450
CPI Error Codes . . . . .	450
General Errors . . . . .	450
File and Pipe Access Errors. . . . .	451
Device Access Errors . . . . .	451
Pipe Routing Service Errors. . . . .	452
Application Service Errors . . . . .	452
Miscellaneous Functions . . . . .	453
<b>Chapter 20. Designing applications with Java . . . . .</b>	<b>455</b>
Setting the classpath in Java 2 . . . . .	455
Setting the classpath in Java 6 . . . . .	456
Using response files to start Java programs on a terminal . . . . .	456
Java application support . . . . .	457
Using Java 6 . . . . .	457
Starting Java 6 . . . . .	458
Properties . . . . .	458
Writing Java 6 Programs . . . . .	459
Access to OS4690 resources and programs. . . . .	460
Internationalization and DBCS enablement in Java . . . . .	460
Locales and encodings in Java . . . . .	461
Fonts in Java 2 . . . . .	461
Installing additional fonts . . . . .	461
Loading DBCS fonts for use by terminal applications . . . . .	462
The font.properties files . . . . .	462
DBCS and Latin character alignment . . . . .	463
Fonts and JavaTOF . . . . .	464
Java limitations on 4690 OS . . . . .	465



File name limitations . . . . .	465
Setting a default directory for Java applications on terminals. . . . .	466
Thread-dispatching differences . . . . .	466
Setting the user's home directory on the terminal . . . . .	466
Java problem resolution . . . . .	467
Frequently asked Java questions. . . . .	472
Additional hints and tips . . . . .	472
Direct serial and parallel port communication . . . . .	474
Java Application Programming Interfaces . . . . .	478
Class FlexosException . . . . .	478
Class InvalidParameterException. . . . .	478
Class POSFile . . . . .	479
Class KeyedFile . . . . .	488
Class POSPipeInputStream. . . . .	496
Class POSPipeOutputStream . . . . .	499
Class ControllerApplicationServices . . . . .	502
Class ControllerStatusData . . . . .	513
Class multiapp.MultiAppStartup . . . . .	520
Class multiapp.PropertyFileApplicationStartup . . . . .	522
Class multiapp.ApplicationManager . . . . .	522
Class multiapp.Application . . . . .	524
Class multiapp.DisplayInfo . . . . .	524
Interface multiapp.ApplicationStartup . . . . .	525
Interface multiapp.ApplicationListener . . . . .	526
Class multiapp.ApplicationEvent . . . . .	526
Class TerminalApplicationServices . . . . .	527
Switching the terminal Java keystroke destination for shared ANPOS keyboards . . . . .	529
Switching the terminal Java, terminal application, and enhanced mode graphical extensions screens . . . . .	530
Class TerminalStatusData . . . . .	530
Class POSPlatform . . . . .	539
Using I/O Redirection to Java . . . . .	539
Handlers and Monitors . . . . .	539
Configuring Redirected Devices . . . . .	540
Class DeviceManager . . . . .	540
DeviceManager Methods. . . . .	541
DeviceManager Tracing . . . . .	542
Interface DeviceManagerInterface . . . . .	543
DeviceManagerInterface Methods . . . . .	544
addDMDeviceRegistrationCompleteListener() . . . . .	544
setJavaInvocationHandler() . . . . .	544
Interface DeviceManagerInterfaceMultiJVM . . . . .	544
DeviceManagerInterfaceMultiJVM Methods . . . . .	544
enableMultiJVM() . . . . .	544
Interface DeviceManagerInterfaceTSS . . . . .	545
DeviceManagerInterfaceTSS Methods . . . . .	545
dispose() . . . . .	545
Interface DMDeviceRegistrationCompleteListener. . . . .	545
DMDeviceRegistrationCompleteListener Methods. . . . .	545
dispose() . . . . .	545
Interface JavaInvocationHandler . . . . .	545
JavaInvocationHandler Methods . . . . .	546
invokeMethod() . . . . .	546
Class JavaInvocationResult. . . . .	546
JavaInvocationResult Constructors . . . . .	546

JavaInvocationResult(Object returnValue).	546
JavaInvocationResult(Throwable exception).	546
Class TerminalKeyboardLights.	547
TerminalKeyboardLights Methods.	547
Class ANDisplayHandler.	548
ANDisplayHandler Methods.	549
Class IOPHandler.	550
IOPHandler Methods.	550
Class ExtendedIOPHandler.	553
ExtendedIOPHandler Methods.	553
Class CashReceiptMonitor.	554
CashReceiptMonitor Method.	554
Class MSRHandler.	554
MSRHandler Methods.	554
Class POSPrinterHandler.	555
POSPrinterHandler Methods.	555
Errors and Exceptions.	557
Class ANDisplayHandlerException.	557
Class IOPHandlerException.	558
Class MSRHandlerException.	560
Class POSPrinterHandlerException.	562
Handler Implementation.	565
Java I/O Processor Functions.	586
Definitions and Concepts.	587
Functions that the Java GUI Performs.	587
JIOP Construction.	587
Sending Input to JIOP.	587
Receiving Events from JIOP.	587
JIOP Configuration.	590
JIOP code example.	591
Java I/O processor packages.	592
Classes in package com.ibm.OS4690.jiop.	593
Class JIOPProcessor.	593
JIOPProcessor Constructor.	593
JIOPProcessor Methods.	593
Class FieldActivatedEvent.	597
FieldActivatedEvent Methods.	597
Class FieldDeactivatedEvent.	598
FieldDeactivatedEvent Methods.	598
Class FieldDataUpdatedEvent.	598
FieldDataUpdatedEvent Methods.	598
Class FullScreenAttributes.	600
FullScreenAttributes Methods.	600
Class IOPReadyListener.	602
IOPReadyListener Methods.	602
Class IOPReadyEvent.	603
Class IOPInitStatus.	604
IOPInitStatus Method.	604
Class PromptChangeListener.	605
PromptChangeListener Method.	605
Class PromptChangeEvent.	607
PromptChangeEvent Methods.	607
Class IOPShutdownListener.	608
IOPShutdownListener Method.	608
Class SystemBusyListener.	609
SystemBusyListener Method.	609

Class SystemBusyEvent . . . . .	610
SystemBusyEvent Method . . . . .	610
Class IOPInputQueue . . . . .	611
IOPInputQueue Methods . . . . .	611
Class QueueStatusChangeListener . . . . .	615
QueueStatusChangeListener Method . . . . .	615
Class QueueStatusChangeEvent . . . . .	616
QueueStatusChangeEvent Method . . . . .	616
Class VideoParms . . . . .	616
VideoParms Method . . . . .	616
IOPInputQueue Public Data . . . . .	619
Interface InputFieldListener . . . . .	621
InputFieldListener Methods . . . . .	621
Interface FullScreenFieldListener . . . . .	621
FullScreenFieldListener Methods . . . . .	621
Interface POSKeyListener . . . . .	622
POSKeyListener Methods . . . . .	622
Interface POSKeyListenerEx . . . . .	623
POSKeyListenerEx Methods . . . . .	623
Interface LabelInputListener . . . . .	624
LabelInputListener Methods . . . . .	624
Class FunctionCodeList . . . . .	625
FunctionCodeList Methods . . . . .	625
Class LabelInputEvent . . . . .	627
LabelInputEvent Methods . . . . .	627
Class QueueLockedException . . . . .	628
Class StateForbidsInputException . . . . .	628
Class StateTableProcessor . . . . .	629
StateTableProcessor Methods . . . . .	629
Interface StateChangeListener . . . . .	633
StateChangeListener Method . . . . .	633
Class StateChangeEvent . . . . .	634
StateChangeEvent Methods . . . . .	634
Class State . . . . .	635
State Methods . . . . .	635
Class FunctionCode . . . . .	639
FunctionCode Methods . . . . .	639
Class Globals . . . . .	646
Globals Method . . . . .	646
Class InputSequenceFormatter . . . . .	647
InputSequenceFormatter Methods . . . . .	647
Class InputSequenceClearedListener . . . . .	649
InputSequenceClearedListener Method . . . . .	649
Class InputSequenceClearedEvent . . . . .	650
InputSequenceClearedEvent Method . . . . .	650
Classes in package com.ibm.OS4690.jiop.util . . . . .	650
Class SystemMonitor . . . . .	651
Class ExceptionListener . . . . .	652
Class ExceptionEvent . . . . .	652
Redirecting Standard Output from the Terminal . . . . .	653
Using an ANPOS Keyboard with Java . . . . .	653
Port 5-Attached ANPOS Keyboard . . . . .	654
PS/2-Attached ANPOS Keyboard . . . . .	654
USB-Attached ANPOS Keyboard . . . . .	654
Using an ANPOS Keyboard with Java IO Processor . . . . .	654
Capturing a Java IO Process Trace . . . . .	654

Capturing a Trace Dump File . . . . .	656
<b>Chapter 21. DBCS support for Java 2 . . . . .</b>	<b>657</b>
Sample DBCS program . . . . .	657
Input Method Operation . . . . .	658
Select and Activate an Input Method (IME) . . . . .	658
Common Features of Input Methods (IME) . . . . .	659
Simplified Chinese Input Method . . . . .	659
Traditional Chinese Input Method. . . . .	660
Japanese Input Method . . . . .	661
Korean Input Method . . . . .	664
<b>Chapter 22. Creating 32-Bit Programs Using VisualAge C/C++ . . . . .</b>	<b>665</b>
Development Platform . . . . .	665
Files Provided for Development . . . . .	665
Support DLLs Provided by the Operating System. . . . .	666
Compiling Your Program . . . . .	666
Linking Your Applications. . . . .	667
Optional Link Libraries. . . . .	667
Wildcard Expansion and Multi-byte Argument Support . . . . .	667
Messages . . . . .	668
Creating DLLs. . . . .	668
C Run-time Environment Variables on 4690 . . . . .	668
Environment File Processing . . . . .	669
Overrides for Specific Machine IDs or Programs . . . . .	670
Special Commands. . . . .	670
Special Environment Variable Settings . . . . .	671
Disabling Buffering of Standard Streams . . . . .	671
Using Locales and iconv functions . . . . .	671
Using the iconv functions. . . . .	672
setlocale and LCL files . . . . .	672
Running Your Application. . . . .	672
Run-time Exception Handling . . . . .	672
JNI-Specific Information . . . . .	673
Restrictions, Limitations, and Differences . . . . .	674
Platform Restrictions . . . . .	674
Differences Between Windows NT and 4690 OS . . . . .	674
4690-Specific Behavior . . . . .	676
Terminal Differences . . . . .	676
Using JNI in 4690 . . . . .	677
Unsupported Features. . . . .	677
Problem resolution . . . . .	678
<b>Chapter 23. Using the Enhanced Security Passwords API . . . . .</b>	<b>679</b>
Introduction to enhanced passwords . . . . .	679
User ID rules . . . . .	679
Password rules . . . . .	679
Authorization attributes . . . . .	680
User-defined attributes . . . . .	681
Error codes. . . . .	681
Sessions. . . . .	683
Application Programming Interfaces. . . . .	684
IsAvailable . . . . .	685
GetCurrentUserID . . . . .	685
Validate . . . . .	686
ValidateID . . . . .	686

ChangePassword . . . . .	687
GetAllAttributes . . . . .	687
GetAllAttributesCurrentUser . . . . .	688
StartSession . . . . .	689
StartSessionCurrentUser . . . . .	690
GetAttributes . . . . .	690
GetAttribute . . . . .	691
GetUserAttributes . . . . .	692
Lock . . . . .	692
Copy . . . . .	693
SetAttributes . . . . .	694
SetAttribute . . . . .	695
SetUserAttributes . . . . .	696
GetGroupNumber . . . . .	697
SetGroupNumber . . . . .	697
GetUserNumber . . . . .	698
SetUserNumber . . . . .	698
GetAuthorizationLevel . . . . .	699
SetAuthorizationLevel . . . . .	700
SetExpires . . . . .	700
Create . . . . .	701
Rename . . . . .	702
Delete . . . . .	702
SetPassword . . . . .	703
GetRecordCount . . . . .	704
GetIDs . . . . .	704
GetModelCount . . . . .	705
GetModelIDs . . . . .	705
GetExpires . . . . .	706
SetExpires . . . . .	706
GetMinPasswordLength . . . . .	707
SetMinPasswordLength . . . . .	707
GetExpireWarning . . . . .	708
SetExpireWarning . . . . .	708
GetMinChange . . . . .	709
SetMinChange . . . . .	709
GetMultipleCharacter . . . . .	710
SetMultipleCharacter . . . . .	711
Commit . . . . .	711
Cancel . . . . .	712
<b>Chapter 24. Designing Python Programs for 4690 . . . . .</b>	<b>713</b>
Python version information . . . . .	713
Using Python on 4690 . . . . .	713
Command mode . . . . .	713
Background Application . . . . .	713
Other Launch Mechanisms . . . . .	714
4690 OS Terminal Support Statement . . . . .	714
Standard Library Functions . . . . .	715
Working with Files and External Programs on 4690 . . . . .	721
Running programs on 4690 . . . . .	721
Running embedded linux programs . . . . .	722
Differences Between Console and Graphical Output . . . . .	724
Installing Third Party Packages . . . . .	724
Limitations . . . . .	726

<b>Appendix A. Operating system disk directory</b>	727
Naming conventions for operating system files	727
Dictionary of operating system files	727
<b>Appendix B. Error Messages</b>	731
LIB86 Error Messages	731
POSTLINK Error Messages	732
LINK86 Error Messages	734
Object File Error Codes	738
<b>Appendix C. Character Sets and Check Printing Application</b>	741
Example of a Normal Width Character Set	741
Example of a Double Width Character Set	742
Example Application for Printing Checks on the Model 2 Printer	742
<b>Appendix D. Notices</b>	751
Telecommunication regulatory statement	752
Electronic Emission Notices	752
Federal Communications Commission (FCC) Statement	752
Industry Canada Class A Emission Compliance statement	752
Avis de conformité à la réglementation d'Industrie Canada	752
Australia and New Zealand Class A Statement	752
European Union Electromagnetic Compatibility (EMC) Directive	
Conformance Statement	752
Germany Class A Statement	753
Japan Voluntary Control Council for Interference Class A statement	754
Japan Electronics and Information Technology Industries Association (JEITA)	
statement	754
Korean communications statement	754
Russian Electromagnetic Interference (EMI) Class A statement	754
People's Republic of China Class A electronic emission Statement	755
Taiwan Class A compliance statement	755
European Community (EC) Mark of Conformity Statement	756
Electrostatic Discharge (ESD)	756
Japanese Electrical Appliance and Material Safety Law statement	756
Japanese power line harmonics compliance statement	756
Cable ferrite requirement	756
Product recycling and disposal	757
Battery return program	758
For Taiwan:	758
For the European Union:	758
For California:	759
Flat panel displays	759
Monitors and workstations	759
Trademarks	760
<b>Glossary</b>	761
<b>Index</b>	781

---

## Figures

1. Example of a random file . . . . .	11
2. Example of a direct file . . . . .	12
3. Example of a keyed file . . . . .	13
4. Example of a sequential file . . . . .	15
5. Multi-Track Reader Data Formats (except JUCC MSR) . . . . .	87
6. Multi-Track Reader Data Formats (including JUCC MSR) . . . . .	88
7. Exceeding the Journal Buffer Example . . . . .	112
8. Folding Algorithm . . . . .	186
9. Example of Home Record Add without Overflow Present. . . . .	192
10. Example of Home Record Add with Overflow Present . . . . .	192
11. Example of Adding a Record to a Full Home Block Creating Overflow Chain . . . . .	193
12. Example of Home Record Add with Overflow Expulsion . . . . .	193
13. LINK86 Operation . . . . .	208
14. Using Overlays in a Large Program . . . . .	217
15. Tree Structure of Overlays . . . . .	217
16. 14-Byte Buffsize . . . . .	340
17. 18-Byte Buffsize . . . . .	341
18. 2x20 Display Areas Example . . . . .	606
19. Output Example of DBCS Program. . . . .	658





## Tables

1. Naming files on the Operating System . . . . .	18
2. File Use Table . . . . .	19
3. LDSN and LFN terms . . . . .	20
4. System-provided LDSN format . . . . .	20
5. Defining Customized Characters . . . . .	41
6. APA Display Guidance Indicators . . . . .	43
7. Examples of OFF, ON and DIRECT . . . . .	43
8. Feature A Video Display Format Types . . . . .	47
9. Video Display Format Types . . . . .	48
10. Applicable Formats of Returned Data . . . . .	88
11. Model 3 or 4/4A Printer—Emphasized Print Definition Characters . . . . .	108
12. Model 3 or 4/4A Printer Font Definition Characters . . . . .	109
13. Model 3 or 4/4A Printer—Receipt Paper Cutter Control Characters . . . . .	114
14. Common MICR Errors . . . . .	120
15. 4610 Printer—Receipt Paper Cutter Control Characters . . . . .	125
16. Errors on the store controller and terminal . . . . .	163
17. Errors unique to the terminal . . . . .	164
18. Chain of Blocks in a Keyed File Using Relative Pointers (example) . . . . .	194
19. LIB86 Command-Line Options . . . . .	201
20. LINK86 Command Options and abbreviations . . . . .	209
21. Command-File Option Parameters . . . . .	210
22. Using Time Frame Indicators . . . . .	239
23. Staged IPL Utility Messages . . . . .	246
24. JavaTOF Property File Editor Error Codes . . . . .	257
25. ADXSTART return codes . . . . .	281
26. ADXSTAR return codes . . . . .	282
27. FUNC Parameter for Operator Authorization Return Codes . . . . .	284
28. Function PRSxCRT Two-Byte Integers . . . . .	288
29. Function PRSxWRT Four-Byte Integers . . . . .	290
30. PRSCWRC function four-byte integers . . . . .	291
31. ADXSERVE return codes . . . . .	294
32. Store controller application status . . . . .	296
33. Terminal application status . . . . .	297
34. ADXCOPYF return codes . . . . .	316
35. Successful ADXDIR return codes that do not imply error conditions. . . . .	322
36. ADXDIR error return codes that imply error conditions. . . . .	323
37. Return code 0 string formatting . . . . .	323
38. Return code 1 string format . . . . .	323
39. Return code 0 string format . . . . .	323
40. Return code 1 string format . . . . .	324
41. Function return codes . . . . .	331
42. Pipe Routing Services Return Codes . . . . .	363
43. Return Codes for Conditional Write to a Pipe Routing Services Pipe . . . . .	364
44. Function Return Codes . . . . .	366
45. Exception Log Entries Returned in User Buffer . . . . .	374
46. Delete Master Exception Log Entries DATABUFF . . . . .	379
47. Delete File Server Exception Log Entries DATABUFF . . . . .	380
48. Function Return Codes . . . . .	381
49. Error Codes . . . . .	383
50. Unique Error Codes . . . . .	388
51. Maximum Length Values for Devices . . . . .	426
52. Java problem resolution . . . . .	467
53. Defining function codes for keys when using an ANPOS keyboard with the Java IO Processor . . . . .	654

54. IMEs and File Names . . . . .	658
55. SCIM Special Keys and Functions . . . . .	660
56. TCIM Special Keys and Functions . . . . .	661
57. Keyboard Mapping Keys and Function . . . . .	662
58. Character Size Keys and Function . . . . .	662
59. Romaji-to-Kana Toggle Key and Function . . . . .	662
60. Keys for Composing a Kanji String . . . . .	663
61. Korean Conversion Key and Function. . . . .	664
62. Problem resolution. . . . .	678
63. Authorization attributes . . . . .	680
64. Python Standard Library Functions. . . . .	715
65. Dictionary of predefined subdirectories and their contents . . . . .	727

# Safety

Before installing this product, read Safety Information- Read This First.

قبل تركيب هذا المنتج، يجب قراءة الملاحظات الأمنية

Antes de instalar este produto, leia as Informações de Segurança.

在安裝本產品之前，請仔細閱讀 **Safety Information**  
(安全信息)。

安裝本產品之前，請先閱讀「安全資訊」。

Prije instalacije ovog produkta obavezno pročitajte Sigurnosne Upute.

Před instalací tohoto produktu si přečtěte příručku bezpečnostních instrukcí.

Læs sikkerhedsforskrifterne, før du installerer dette produkt.

Lees voordat u dit product installeert eerst de veiligheidsvoorschriften.

Ennen kuin asennat tämän tuotteen, lue turvaohjeet kohdasta Safety Information.

Avant d'installer ce produit, lisez les consignes de sécurité.

Vor der Installation dieses Produkts die Sicherheitshinweise lesen.

Πριν εγκαταστήσετε το προϊόν αυτό, διαβάστε τις πληροφορίες ασφάλειας  
(safety information).

לפני שתתקינו מוצר זה, קראו את הוראות הבטיחות.

A termék telepítése előtt olvassa el a Biztonsági előírásokat!

Prima di installare questo prodotto, leggere le Informazioni sulla Sicurezza.

製品の設置の前に、安全情報をお読みください。

본 제품을 설치하기 전에 안전 정보를 읽으십시오.

Пред да се инсталира овој продукт, прочитајте информацијата за безбедност.

Les sikkerhetsinformasjonen (Safety Information) før du installerer dette produktet.

Przed zainstalowaniem tego produktu, należy zapoznać się  
z książką "Informacje dotyczące bezpieczeństwa" (Safety Information).

Antes de instalar este produto, leia as Informações sobre Segurança.

Перед установкой продукта прочтите инструкции по  
технике безопасности.

Pred inštaláciou tohto zariadenia si pečítajte Bezpečnostné predpisy.

Pred namestitvijo tega proizvoda preberite Varnostne informacije.

Antes de instalar este producto, lea la información de seguridad.

Läs säkerhetsinformationen innan du installerar den här produkten.

---

## About this guide

This guide describes how to program the 4690 OS Version 6 Release 4 (hereafter referred to as the *operating system*) and the interfaces for application programs.

Throughout this guide, any reference to Toshiba products is also referring to those products matching the Toshiba product description, that were formally labeled as IBM products. References to Toshiba cash drawers includes older cash drawers that were previously labeled as IBM cash drawers. IBM devices are not considered non-Toshiba devices.

---

## Who should read this guide

This guide is written for the programmer who uses system services or writes application programs to run on the operating system.

---

## Terminal models

The 4693-xx1/4694, SurePOS™ 300/700 Series and TCxWave™ 6140 Series terminals are called *Mod1* terminals. Although all are called *Mod1* terminals, each terminal model supports some features that other models do not support. Additionally, the SurePOS 300/700 Series and TCxWave 6140 Series terminals provide Universal Serial Bus (USB) capabilities.

The 4693-xx2 terminals are called *Mod2* terminals. These terminals attach to a *Mod1* terminal and depend upon that *Mod1* terminal for control and communication with the store controller.

| **Note:** 4683 terminals are not supported on 4690 OS V6R3 or later. References to 4683 terminals only  
| apply to previous versions of the OS.

The controller/terminal (for example, a 4693-5x1 controller/terminal) combines the function of the store controller and point-of-sale (POS) terminal in a single product. The terminal portion of a controller/terminal is considered to be a *Mod1* terminal.

**Note:** 4694 and SurePOS 700 Series (Models 72x, 74x, 75x and 78x) systems are always valid as controller/terminals. 4693 systems are only supported as a controller in a non-Java environment or as an alternate in a Java environment.

4690 OS V6R3 introduced support for the SurePOS 300 Series Model 350 terminal (4810-350). This is a *Mod1* terminal. The 4810-350 has the following characteristics:

- It can only be used as a terminal in Enhanced Mode
- It cannot be used as a store controller
- The RS232 Sureport card is not supported by 4690 OS
- There are no RS485 ports available

| 4690 V6R4 introduced support for the TCxWave Series terminal (6140-100). The 6140-100 is a *Mod1*  
| terminal with the following characteristics:

- | • It can only be used as a terminal in Enhanced Mode
- | • It cannot be used as a store controller
- | • No RS485 ports available, including no cash drawer port
- | • Cash drawer support is provided through USB-attached cash drawers
- | • RS232 support is provided through RS232-to-Serial dongles
- | • The Power Button functions as the only available Dump Button

---

## Where to find more information

Current versions of Toshiba publications are available on the Toshiba support site.

1. On the right side of the web page under popular links, select **Publications**.
2. Click on the publication related to your product.

## 4690 V6 library

**Note:** References to related 4690 publications in this guide are references to the publications in the 4690 Version 6 library. For example, the *4690 OS Version 6: Programming Guide* is referred to as the *4690 OS: Programming Guide*.

*4690 OS Version 6: Planning, Installation, and Configuration Guide*, G362-0541

*4690 OS Version 6: User's Guide*, G362-0542

*4690 OS Version 6: Messages Guide*, G362-0543

*4690 OS Version 6: Communications Programming Reference*, G362-0544

*4690 OS Version 6: Programming Guide*, G362-0545

*4690 OS Version 6: Master Index*, G362-0546

*4680 BASIC: Language Reference*, SC30-3356

For the latest information on the TCxWave 6140 Series publications, see the Toshiba support site.

---

## Notice statements

Notices in this guide are defined as follows:

<b>Notes</b>	These notices provide important tips, guidance, or advice.
<b>Important</b>	These notices provide information or advice that might help you avoid inconvenient or problem situations.
<b>Attention</b>	These notices indicate potential damage to programs, devices, or data. An attention notice is placed just before the instruction or situation in which damage could occur.
<b>CAUTION</b>	These statements indicate situations that can be potentially hazardous to you. A caution statement is placed just before the description of a potentially hazardous procedure step or situation.
<b>DANGER</b>	These statements indicate situations that can be potentially lethal or extremely hazardous to you. A danger statement is placed just before the description of a potentially lethal or extremely hazardous procedure step or situation.

---

## **Part 1. The operating system environment**





---

# Chapter 1. The operating system environment

This chapter is an overview of the operating system environment. It provides the capabilities of the system and refers you to other chapters for more detailed information.

---

## Operating modes

4690 OS V6 introduces a new infrastructure laying the way for the future while continuing to provide the expected benefits of the OS. As a result, 4690 OS V6 provides two operating modes, Classic and Enhanced.

### Classic Mode

Classic Mode continues to use the OS infrastructure on which previous releases were based. This mode is intended to allow 4690 OS V6 to support the functions and much of the same hardware that have been supported in previous 4690 versions. Some new hardware and new functions available in 4690 OS V6 are not supported in Classic Mode. Other functions, such as the new RMA software distribution support, is available in both modes.

Some 4690 OS V6 functions are available only in Classic Mode. These include, but are not limited to:

- APM Standby
- Tape backup devices
- Optical drives (O:)
- Display/Alter - physical mode
- Disable Write Verify function

### Enhanced Mode

Enhanced Mode introduces a new hardware interface layer below Classic 4690 OS. The current 4690 OS user interfaces and programming APIs continue to be available allowing current applications to run in Enhanced Mode. This mode supports some additional IBM System x servers not supported by Classic Mode, including selected IBM blade server models. Enhanced Mode is also required in order to exploit other new functions such as USB flash memory drives.

Systems running supported versions of 4690 OS that meet the minimum resource requirements of 4690 OS V6 may be migrated to 4690 OS V6 and continue to run in Classic Mode. The 4690 OS V6 installation CD is only available as an Enhanced Mode installation. 4690 OS V6 provides support for some System x servers only in Enhanced Mode. See Appendix L of the Planning, Installation, and Configuration Guide, "Classic To Enhanced Conversion Utilities," for information on converting controllers from Classic Mode to Enhanced Mode.

Some new functions in 4690 OS V6 are available only in Enhanced Mode. These include, but are not limited to:

- Booting Supplemental system from hard disk
- USB memory key support
- Creating Supplementals on CD or USB memory key
- Java 1.6 support on the controller
- F: drive for Java 1.6
- RMA Master Agent on 4690 OS controller
- Enhanced Options menu
- S3-Deep Sleep Support
- WOL-Wake On LAN Support

- Directory Services

---

## System capabilities

The operating system is a multitasking and multiuser environment that runs in the store controller, the point-of-sale terminals, and the controller/terminals. The operating system can handle store controller communications with point-of-sale terminals, other controllers, the controller/terminal, and the host processor. The following sections briefly describe the system's capabilities.

Beginning with 4690 OS Version 6 Release 4 Token Ring communications is no longer supported.

---

## Concurrent operations

The following functions of the operating system allow concurrent operations:

- The operating system allows you to run your batch and interactive programs at the same time. One program does not have to finish before another program can start.
- The operating system allows several operators to use the system at the same time. Your operators can be authorized to run one program at a time or several at a time. Operators are authorized through the *system authorization file*. See Chapter 16, "Designing applications with 4680 BASIC," on page 277 for information on how to authorize users on the system.
- Multiple users on the system can run multiple interactive applications through the use of *windows*. An operator can switch from window to window, start or stop an application running on a window, or check the status of an application running on a window. Windows are possible through a window control facility. See the *4690 OS: User's Guide* for an explanation on how windows are used on the system.
- You can communicate with a host site at the same time your in-store communications are taking place.
- You can attach multiple serial devices to your store controller. The Multiple Console facility handles the management of these devices as additional system consoles.

---

## Operator interface

For some system functions (for example, configuration) you communicate with the system through a menu-driven interface at the store controller console. For other functions (for example, copying files) you enter commands directly into the system. You can attach auxiliary consoles to the store controller. For information on how to use system functions and auxiliary consoles, see the *4690 OS: User's Guide*.

---

## Changing the signon screen

You can change the signon screen displayed during signon by creating an alternate logo in the file C:\ADX\_IPGM\ADXLOGOD.DAT. If this file exists during system startup, the first 10 rows of data on the signon screen are replaced with the first 10 lines of alternate logo data in this file.

ADXLOGOD.DAT should reside in the ADX\_IPGM subdirectory. This file can be applied using Apply Software Maintenance (ASM). See the *4690 OS: User's Guide* for information on the ASM utility program. When using ASM, the Advanced Data Communications Systems (ADCS) Host Command Processor (HCP) six-character name should be ?LOGOD. You maintain the contents and distribution attributes of this file.

The alternate logo can be a maximum of 10 rows with 79 columns in each row. Each row should be terminated with a carriage return and a line feed. The operating system editor DR EDIX automatically terminates lines with these characters.

---

## Programming auto sign off

The Auto Sign Off function allows you to sign off or disconnect a 4690 system console user who has been inactive (no system keyboard or mouse input) for a specifiable length of time. This function is available as a set of 4690 controller application services in 16-bit and 32-bit C, Java, and CBASIC. These services are available only for background applications. They cannot be used from a terminal application or a controller foreground application.

### Wait for system console keyboard and mouse inactivity

This service returns when no controller keyboard, terminal Java keyboard, or mouse activity has occurred on the system console for the time specified.

An operator using 4690 Remote Operator and a client logged on to the 4690 telnet server share the controller keyboard and display with the system console. Therefore, keyboard activity by a remote operator or a telnet client will cause the inactivity timer for the system console to be reset.

Enhanced 4690 telnet clients and physical auxiliary consoles are configured as auxiliary consoles and do not share the controller keyboard and display with the system console; therefore, keyboard activity by an enhanced telnet client or a physical auxiliary console will not cause the inactivity timer for the system console to be reset. Refer to “Programming auto sign off for system and auxiliary consoles” on page 6 to program auto signoff for both the system and auxiliary consoles.

Refer to “Wait for system keyboard and mouse inactivity” on page 308, “Wait for system console keyboard and mouse inactivity” on page 375, and “Wait for system console keyboard and mouse inactivity” on page 375 for information about this service for CBASIC, 16-bit and 32-bit C, and Java.

**Note:** Only one keyboard and mouse wait (either activity or inactivity) can be outstanding at a time.

### Wait for system console keyboard and mouse activity

This service returns at the first controller keyboard, terminal Java keyboard, or mouse activity on the system console.

An operator using 4690 Remote Operator and a client logged on to the 4690 telnet server share the controller keyboard and display with the system console. Therefore, keyboard activity by a remote operator or a telnet client will cause this function to return.

Enhanced 4690 telnet clients and physical auxiliary consoles are configured as auxiliary consoles and do not share the controller keyboard and display with the system console; therefore, keyboard activity by an enhanced telnet client or a physical auxiliary console will not cause this function to return. Refer to the “Programming auto sign off for system and auxiliary consoles” on page 6 program auto signoff for both the system and auxiliary consoles.

Refer to “Wait for system keyboard and mouse inactivity” on page 308, “Wait for system console keyboard and mouse inactivity” on page 375, and “Wait for system console keyboard and mouse inactivity” on page 375 for information about this service for CBASIC, 16-bit and 32-bit C, and Java.

**Note:** Only one keyboard and mouse wait (either activity or inactivity) can be outstanding at a time.

### Sign off the active user

This service signs off the system console active user. This service is valid only when the controller is in controller or terminal Java mode. Refer to “Sign off the system console active user” on page 308, “Sign off the system console active user” on page 376, and “signoffActiveUser()” on page 510 for information about this service for CBASIC, 16-bit and 32-bit C, and Java.

## Disconnect the active user

This service disconnects the system console active user. This service is valid only when the controller is in controller or terminal Java mode. Refer to “Disconnect the system console active user” on page 308, “Disconnect the system console active user” on page 376, and “disconnectActiveUser()” on page 511 for information about this service for CBASIC, 16-bit and 32-bit C, and Java.

## Example

The following is a simple example of how the Auto Sign Off function application services can be used to auto sign off a user.

```
Begin loop
  Wait for System Console Keyboard and Mouse Inactivity
    completes when keyboard activity occurs to sign on user
  Wait for System Console Keyboard and Mouse Activity
    completes when specified time elapses with no keyboard or mouse input
  Sign off (or Disconnect) the Active User
    active user is signed off (or disconnected) and 4690 Signon Screen is displayed
  Return to top of loop
End loop
```

---

## Programming auto sign off for system and auxiliary consoles

For 4690 Enhanced systems there are additional 4690 application services that allow you to monitor the system console and all configured auxiliary consoles, and to signoff or disconnect active users on those consoles.

These services are:

- Available in 16-bit and 32-bit C, Java, and CBASIC
- Available for background applications only. They cannot be used from a terminal application or a controller foreground application.
- Not available on 4690 Classic systems

## Start inactivity monitoring

The system console and all configured auxiliary consoles will be monitored for inactivity.

Keyboard and mouse activity is monitored for the system console. An operator using 4690 Remote Operator and a client logged on to the 4690 telnet server share the controller keyboard and display with the system console. Therefore, keyboard activity by the remote operator or the telnet client will cause the inactivity timer for the system console to be reset.

Keyboard activity is monitored for the auxiliary consoles. Enhanced 4690 telnet clients and physical auxiliary consoles can be one of the eight auxiliary consoles that can be defined with controller configuration.

If there are no active users on any consoles at the time the start inactivity monitoring call is made, it will return immediately and no consoles will be monitored. If there are active users, the call will block the application until the system and/or an auxiliary console has had inactivity for the requested amount of time.

When the call returns with a console(s), all other consoles continue to be monitored until a stop inactivity monitoring call is made. On a subsequent start inactivity monitoring call, the inactivity timer is not reset on any consoles that have active users. Because of this, it is possible the call will return immediately with consoles that have already timed out.

The application should use this call in a loop to achieve desired results – that is, any active user signed off after a period of inactivity.

## Example

The following is an example of the Auto Sign Off function for system and auxiliary consoles.

A controller with auxiliary consoles 3,4, and 8 configured  
Want to signoff any user who has been inactive for 15 minutes  
Background application in loop ... sample activity ...

```
Start inactivity monitoring with 15 minute timeout
Returns immediately with no active users return code
Repeat - Start inactivity monitoring with 15 minute timeout
Users have signed on to system console and consoles 3 and 4
Since active users, call blocks waiting for inactivity
Users on system console and consoles 3 and 4 keep keyboard busy
Returns with console 8 because no keyboard activity on console 8
Signoff console 8 – get no active user return code
Repeat - Start inactivity monitoring with 15 minute timeout
Since active users, call blocks waiting for inactivity
Users on system console and console 4 have stopped activity for 15 minutes
Returns with system console and console 4.
Signoff system console – success
Signoff console 4 – success
Repeat - Start inactivity monitoring with 15 minute timeout
Since active users, call blocks waiting for inactivity
User on console 3 has stopped activity for 15 minutes
Returns with console 3
Signoff console 3 – success
Start inactivity monitoring with 15 minute timeout
Returns immediately with no active users return code
```

Refer to “Start inactivity monitoring” on page 308 for information about this service for CBASIC, 16-bit and 32-bit C, and Java.

## Stop inactivity monitoring

Inactivity monitoring for the system console and all configured auxiliary consoles will be stopped. Refer to “Stop inactivity monitoring” on page 309 for information about this service for CBASIC, 16-bit and 32-bit C, and Java.

## Signoff an active user

Signoff the active user of a specific console or all consoles. On a controller/terminal, you can signoff the system console active user only when the controller/terminal is in controller mode.

Refer to “Signoff specific or all active users” on page 309 for information about this service for CBASIC, 16-bit and 32-bit C, and Java.

## Disconnect an active user

Disconnect the active user of a specific or all consoles. On a controller/terminal, you can signoff the system console active user only when the controller/terminal is in controller mode.

Refer to “Disconnect specific or all active users” on page 310 for information about this service for CBASIC, 16-bit and 32-bit C, and Java.

---

## Communications

The following list contains communications-related information:

- The communication interfaces that support Systems Network Architecture (SNA) protocols are logical unit (LU) 0 and LU 6.2 as node types 2.0 and 2.1. The line connection supported on the non-SNA interface is asynchronous (ASYNC) Communication. The line connections supported on the SNA

interface are Synchronous Data Link Control (SDLC), token ring, Ethernet, local link, and X.25. The *4690 OS: Communications Programming Reference* explains how to use host and peer communications.

- LU 6.2 allows user-written transaction programs (TPs) on the store controller to communicate with advanced program-to-program communications (APPC) applications on a host node (type 4 or 5) or on a peer node (type 2.1). The TP accesses the LU 6.2 communications functions by calling APPC functions. These functions are known as calls (verbs) from a set of library routines that are linked to the TP. CPI Communication is the LU 6.2 communications interface used by the operating system. The *4690 OS: Communications Programming Reference* explains LU 6.2 communications.
- Application-to-application communications are processed through in-memory files called *pipes*. For information on how pipes work with your applications, see Chapter 16, “Designing applications with 4680 BASIC,” on page 277 and Chapter 17, “Designing Applications with Other Languages,” on page 333.
- You can optionally link your store controllers together by the token ring or Ethernet and they can communicate with each other using the Multiple Controller Feature (MCF). A store controller configured as the master store controller serves as the central point of control on this type of system.
- On a system using the MCF, files are updated and distributed as you specify to other store controllers on the network. For more information, see the *4690 OS: User's Guide*.

---

## Files

This list briefly describes file security, file types, and file structures:

- The operating system provides protection for your files by limiting access to only authorized users. To control file access, you assign each user to one of three categories. See “Protecting files” on page 33 for a description of these categories and their uses.
- You can specify and manage the way disk files are shared with other users. See “Sharing files” on page 31 for information on allowing file access.
- You can create and manage keyed files and other files. A utility allows you to change keyed files into direct files or direct files into keyed files. See “Keyed” on page 12 for information about keyed files and how to change these files into direct files, or direct files into keyed files. “Creating files” on page 29 explains how files are created on the system.
- The system uses a hierarchical file structure similar to the IBM Personal Computer Disk Operating System (IBM DOS).
- The operating system allows the use of file names greater than eight characters in length. See “Using long file names on VFS systems” on page 28 for information on this feature.

---

## Problem-determination aids

The operating system provides the following problem-determination aids:

- The operating system provides several problem-determination aids to help you analyze problems in your programs or on your system. One of the aids allows you to collect data about a problem, then print, display, or file the data. You can also create a diskette for documenting and analyzing a problem. See the *4690 OS: Messages Guide* for information on collecting problem-analysis data.
- A system error log provides a record of system errors that occurred. You can use this data to help identify and resolve problems. For information on the system error log, see Chapter 4, “Using error recovery procedures and facilities,” on page 153.

---

## Utilities

The operating system provides the following utilities:

- A utility is available to help you develop the input sequence table. The input sequence table lets you edit operator input from various terminal input devices. Chapter 7, “Using the Input Sequence Table Build Utility,” on page 197 describes the input sequence table utility.



- The Apply Software Maintenance (ASM) Utility allows you to update system or user programs through a menu-driven interface. See the *4690 OS: User's Guide* for information about this utility. The Staged IPL Utility lets you apply maintenance using ASM when one controller is up and able to run TCC Networks. See Chapter 13, "Using the Staged IPL Utility," on page 241 for information on the Staged IPL Utility.
- Special write operations protect your data in the event of a power line disturbance. For information on recovering from power line disturbances, see Chapter 4, "Using error recovery procedures and facilities," on page 153.
- You can use the Distributed File utility to distribute files to other nodes on a multiple-controller system. See the *4690 OS: User's Guide* for a complete description of this utility.
- Two text editing facilities, DR EDIX and XE Editor, let you create and edit files. See the *4690 OS: User's Guide* for information on the text editing facility.

---

## Additional products

The following list contains information about other products related to the operating system:

- The operating system supports an optional high-level debugging aid called the *4680 Application Debugger*. It is available by RPQ P85154.
- The operating system includes a Display Manager program that enables you to create, modify, or manage information displayed on the store controller screen. See the *4680 Store System: Display Manager User's Guide* for information about this program.

---

## Store controller backup

You specify the *store controller backup* during configuration. This backup automatically backs up your primary controller when activated. The function provides continuous control for your terminal operations. When you configure and prepare a store controller for backup, it can assume control of the TCC Network when the store controller for that TCC Network is not operational. For information on configuring and preparing store controller backup, see the *4690 OS: User's Guide*.

The operating system also provides data backup. A multiple-controller system provides *Data Backup*, the capability for file redundancy. This capability enables store controllers to communicate with each other across the local area network (LAN). With this feature you can specify when your files are to be automatically updated and distributed. For more information on data backup with a LAN, see the *4690 OS: User's Guide*.





## Chapter 2. Managing files

This chapter shows you how to handle files. It gives you information about file types and describes how to manage your files and subdirectories.

### Selecting file types

You can choose from four types of files when building the files for your store system: random, direct, keyed, and sequential. You create all files using variations of the CREATE statement and accompanying parameters.

#### Random

Random files have fixed record lengths. The system uses the last two bytes in each record for the carriage return and line feed characters (CR/LF). The system pads the data space between the last field of the record and the CR/LF.

Random files offer random access, which allows direct access to any record in a file. This ability makes random and direct files an ideal type for files that can be referenced by a relative record number. If you need a name or a specific number (such as a telephone number) to reference a record, use Keyed File Services. You ensure that the number of bytes occupied by the field delimiters and CR/LF do not exceed the specified record length. Figure 1 on page 11 shows a random file composed of three records. The data is in ASCII characters.

FILE.1	RECORD 1	"FIELD ONE","FIELD TWO","FIELD THREE" CR/LF
	RECORD 2	"FIELD 1","FIELD 2"," " CR/LF
	RECORD 3	111,222,3.3,444,5.5 CR/LF
		← Record lengths fixed →

Figure 1. Example of a random file

4680 BASIC locates each record of a randomly accessed file by taking the record number, subtracting 1, and then multiplying that result by the record length. The result is a byte displacement value for the record measured from the beginning of the file. Records can span sectors. You must specify the record to be accessed in each READ#, PRINT#, or WRITE# statement executed. The following BASIC program creates the random file diagrammed in Figure 1 on page 11.

```
CREATE "FILE.1" RECL 40 AS 2
  A$ = "FIELD ONE"
  B$ = "FIELD TWO"
  C$ = "FIELD THREE"
  D$ = "Field 1"
  E$ = "Field 2"
  F$ = ""
  G% = 111
  H% = 222
  I  = 3.3
  J% = 444
  K  = 5.5
WRITE #2,1; A$, B$, C$
```

```

WRITE #2,2; D$, E$, F$
WRITE #2,3; G%, H%, I, J%, K
CLOSE 2
END

```

The following program reads the first three fields from record 3 in FILE.1.

```

IF END #20 THEN 200
OPEN "FILE.1" RECL 40 AS 20
  READ #20,3; FIELD1%, FIELD2%, FIELD3
  PRINT FIELD1%, FIELD2%, FIELD3
200 END

```

The preceding program results in the following output.

```

111                222                3.3

```

For applications written for the terminal, you must use the WRITE statement to send data to a random file.

## Direct

Direct files are like random files except that direct files contain no delimiting characters (quotes, commas, and CR/LF). Direct files contain no data formatting information. Because of this, you must specify a special format string when you want to access the data in a direct file. See “Performing file functions” on page 29 to learn more about accessing a direct file.

The following program creates a direct file named DIRECT.DAT that consists of one 15-byte record. The variable D represents the format string used in the WRITE FORM# statement.

```

STRING A, D
INTEGER*2 B
REAL C

CREATE "DIRECT.DAT" DIRECT RECL 15 AS 3 LOCKED
  A = "ABC"
  B = 32767
  C = 27.35
  D = "C3, I2, R"
WRITE FORM D; #3,1; A,B,C

CLOSE 3
END

```

Figure 2 on page 12 shows the direct file created by the preceding program. The data in the figure is in hexadecimal.

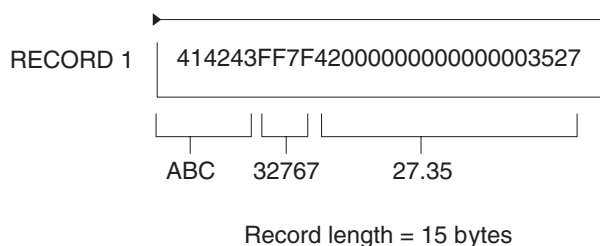


Figure 2. Example of a direct file

## Keyed

Keyed files are composed of fixed-length records that are accessed by using a key, which can be any combination of characters that can be used in a 4680 BASIC string.

A keyed file consists of blocks that contain keyed records. A block is a 512-byte sector. Each block uses the first four bytes for chain pointers and the next 508 bytes for records. A block can contain multiple records, but records are not split across blocks. Each record contains its key and data, with the key always being first. A record can be a maximum of 508 bytes. The key can be as many bytes of the record as you need.

You should use keyed files for files accessed by a name or a number. Examples of such numbers are a Universal Product Code (UPC), a label, a telephone number, or a check authorization number. These types of names and numbers do not have any contiguous nature and are random in their occurrence. You can use numbers that have a contiguous nature to access a random or direct file.

You access keyed files by hashing the key to obtain a relative position within the file to locate the record. Keyed files do not have indexes. The more unique each key is with respect to all of the other keys used in a keyed file, the better the hashing technique works.

To access a keyed file sequentially, use the keyed file as a direct file. You can obtain information about the keyed file from the first block, which is used only for control information. The format of this first block is:

Relative byte offset (in decimal)	Number of bytes (in decimal)	Keyed file information
42	4	Number of blocks in keyed file
46	2	Keyed record size
48	4	Randomizing divisor
54	2	Key length
56	4	Chaining threshold

You can create keyed files directly with a 4680 BASIC program, or you can use the Keyed File Utility program provided with the operating system. The utility provides an efficient two-pass method of converting a direct file into a keyed file. See Chapter 6, "Using the Keyed File Utility," on page 177 for an explanation of this utility.

Like direct files, keyed files use no delimiting characters. You use a format string to map the data to and from the record. Figure 3 on page 13 shows a keyed file record with a 19-byte length. The data in the figure is in hexadecimal.

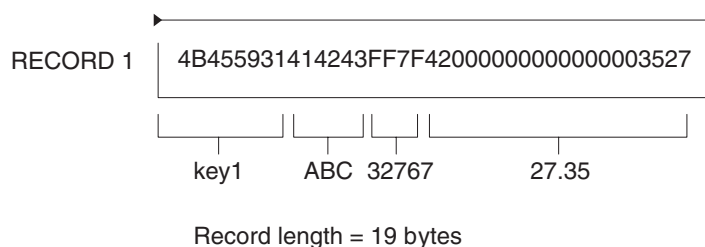


Figure 3. Example of a keyed file

The following program creates the 19-byte keyed file shown in the Figure 3 on page 13. The keyed file consists of one 19-byte record.

```

STRING A, D, KEY
INTEGER B
REAL C

CREATE POSFILE "KEYFILE.DAT" KEYED 4 , , , 200 \
  RECL 19 AS 4 LOCKED
  KEY = "key1"

```

```

A = "ABC"
B = 32767
C = 27.35
D = "C4, C3, I2, R"
WRITE FORM D; #4; KEY,A,B,C

CLOSE 4
END

```

In this program:

- The numeral 4 after the reserved word KEYED specifies a key length of 4 bytes.
- The three commas after the numeral 4 indicate that no randomizing divisor or chaining threshold value is specified; therefore, the defaults are used.
- The numeral 200 after the three commas specifies a total of 200 records in the keyed file.
- The variable D represents the format string used in the WRITE FORM# statement.

## Sequential

Sequential files are composed of variable-length records accessed in a first-record-to-last-record sequence. A record in a sequential file consists of one or more fields delimited by commas between the fields. A CR/LF follows the last field. A string data field is also delimited by quotation marks. A numeric data field is converted to an ASCII representation.

A record in a sequential file can contain up to 64 KB. Individual record lengths vary according to the size of the fields in the record. Each field occupies only the number of bytes required by the data in the field and the delimiters. Data records are written one after another and can span disk sectors.

You can open sequential files for reading or writing but not for both. You can, however, use two OPEN statements to access a sequential file, one for reading and one for writing. Each open processes the file sequentially. When you open a sequential file for reading, you start at the beginning of the file. When you open a sequential file for writing, you must specify the APPEND option in the OPEN statement. The data that you write is appended to the end of the existing file.

Although sequential files are normally accessed from beginning to end, 4680 BASIC provides a way to re-establish a reference within a sequential file for reprocessing a sequential file from a saved reference. You might find this useful as a checkpoint when processing large sequential files. After re-establishing a sequential file reference, the file access is again a next-record sequence. Note that you cannot re-establish a reference when writing to a sequential file.

A special form of the WRITE command, WRITE MATRIX, enables you to write a record to a sequential file from a terminal application. The WRITE MATRIX command supports writing records longer than 512 bytes. It also supports the simultaneous writing of records from more than one terminal. WRITE MATRIX uses several file writes to place all the data in the record in the sequential file. The first write specifies some of the fields and filler data for the remainder of the record. After the first write, the record contains the fields that fit into 512 bytes, and the remainder of the fields are empty (the record is filled with commas and a terminating CR/LF). The other writes fill the remainder of the fields in increments of 512 or fewer bytes. The entire record is locked until all of the writes are finished. The record is unlocked if the connection between the store controller and the terminal is lost. The system writes a WRITE MATRIX record this way to keep an application from reading the record until it is complete. It also allows an application to determine whether the record has been completed or not. Records placed in a sequential file by WRITE MATRIX are not guaranteed to be in the sequence in which they occur.

You should use sequential files for data that is collected in a serial manner. Examples are the sales data collected in the terminal that is associated with each sales transaction, a listing of events as they occur, and data to be saved in a file to be printed later.

Figure 4 on page 15 shows a sequential file composed of three records. The data in the figure is in hexadecimal.

FILE.2	RECORD 1	"FIELD ONE","FIELD TWO","FIELD THREE"CR/LF
	RECORD 2	"FIELD 1","FIELD 2"," "CR/LF
	RECORD 3	111,222,3.3,444,5.5CR/LF

← Record lengths vary →

Figure 4. Example of a sequential file

The third field in RECORD 2 is a null string. Commas and quotation marks serve as delimiters but may also appear in a field as long as they are imbedded within quotation marks. The only exception is that a quotation mark followed by a comma must serve as a string delimiter. The following program creates the sequential file diagrammed in Figure 4 on page 15.

```

CREATE "FILE.2" AS 2
  A$ = "FIELD ZERO, ONE"
  B$ = "FIELD TWO"
  C$ = "FIELD THREE"
  D$ = "Field 1"
  E$ = "Field 2"
  F$ = ""
  G% = 111
  H% = 222
  I = 3.3
  J% = 444
  K = 5.5
WRITE #2; A$, B$, C$
WRITE #2; D$, E$, F$
WRITE #2; G%, H%, I, J%, K
CLOSE 2
END

```

The three WRITE# statements correspond to the three records, and each variable corresponds to a field.

When you access sequential files, each field is read one at a time from the first to the last. The READ# statement considers a field complete when it encounters a comma or CR/LF for numeric fields, and a quotation mark followed by a comma or carriage return for string fields.

The following program reads the fields in FILE.2 sequentially and prints them on the display device, one field for each line.

```

IF END #19 THEN 100
OPEN "FILE.2" AS 19
  FOR I% = 1 TO 11
    READ #19; FIELD$
    PRINT FIELD$
  NEXT I%
100 END

```

The data type should match the variable type on a field-by-field basis.

The preceding program results in the following output:

```

FIELD ONE
FIELD TWO
FIELD THREE

```

Field 1  
Field 2

111  
222  
3.3  
444  
5.5

**Note:** Applications written for the terminal can send data to a sequential file using the WRITE MATRIX statement only.

---

## Naming files and subdirectories

A complete name for a file contains the name of the file, the disk or diskette drive or CD-ROM drive that contains the file, the subdirectory path for the file, and the node name if you have MCF and want to access a file on another store controller. Some of the information in the file name is optional. If it is not present, the system either treats it as blanks or substitutes default values. Generally, you should specify values rather than rely on defaults.

The general format for a complete file name is:

*nodename::drivename:/subdirectory/filename.extension*

where:

### **nodename**

The name used to identify a store controller on the network. You can use this name to access files on a different store controller on the LAN. The *nodename* must be followed by two colons (::). See “Using node names to access files” on page 27 for more information.

### **drivename**

The name for the disk or diskette drive or CD-ROM drive. A is for the first diskette drive. B is for the second diskette drive. C is for the first hard disk drive. D is for the second hard disk drive. P is for the CD-ROM drive. The *drivename* must be followed by one colon (:).

### **subdirectory**

The names of the subdirectories in the path for this file. See Appendix A, “Operating system disk directory,” on page 727 for determining the use of these names. Subdirectory names can contain one to eight characters and are delimited by slashes. You can specify multiple subdirectory names. If you do not specify a subdirectory name, the system uses the root directory. The operating system generally uses the first level of the subdirectory and not the root.

**Note:** On a virtual file system (VFS) system, the maximum path length, including directories and files, is 260 characters. The maximum directory depth is 60 levels including the root directory.

### **filename**

The actual name of the file. The file name can contain one to eight characters on a file allocation table (FAT) system or up to 256 characters on a VFS system.

### **extension**

The extension of the file name. You use the extension as an additional set of characters to uniquely define a file. The extension can contain one to three characters and is separated by a period from the file name.

In an application program, you should use a *logical name* to represent the complete file name. The logical name is easier to use and allows the file to be placed on the appropriate drive without changing the application program. See “Using logical names” on page 20 for information on logical names.

---

## Rules for naming subdirectories and files on FAT systems

The general rules for naming subdirectories and files on FAT systems are:

1. Do not start a subdirectory or file name with ADX. The prefix ADX is reserved for the operating system files.
2. Avoid using a name for a file that matches a name of an operating system command.
3. Characters allowed in subdirectories, file names, and extensions are:
  - Uppercase A–Z
  - 0–9
  - Special characters @ # ( ) { } \$ &

**Note:** Do not use these special characters in the **first, fourth, fifth, or eighth** position of a file name because these conflict with Host Command Processor (HCP) and input sequence table build file names.

4. The following characters are not allowed in subdirectories, file names, and extensions:  
? \* : . ; , [ ] ! + = < > " - / \ |
5. In terminal applications, you can use a maximum of 24 characters in a statement for a file name, including its extension. This maximum includes a mandatory node name (for example, R::) of three characters prior to all file names.
6. In the store controller, you can use a maximum of 127 characters in a statement for a file name, including its extension.
7. File names with a .BSX extension are symbol files for programs with the same file name.
8. The file extension .\$\$\$ represents a temporary file that the data distribution application (DDA) uses when distributing an entire file across the LAN (MCF Network).  
For example, when distributing the file ABC.DAT to a remote node, the application creates a new version named ABC.\*\*\*. The application deletes the ABC.DAT copy and renames the temporary file (ABC.\*\*\* ) to ABC.DAT.
9. The file extension .C0 is a temporary input file to the Keyed File Utility.
10. All ADXxxxx names are not files or subdirectories. Some ADX file names represent functions, pipes, processes, or node names. For example:

### **ADXFILES**

Function for a forced close

### **ADXLNDAP**

Pipe indicating that a store controller is the acting master

### **ADXLND1L**

LAN distribute per update process

### **ADXLXCCN::**

LAN node name for store controller CC

### **ADXLXAAN::**

LAN node name for acting master store controller

11. All operating system files are found in directories. Some files are placed in the root directory, but most are located in subdirectories that are one level below the root directory.
12. Operating system subdirectory names have the following general form:

ADX\_yzzz

Where:

**y** = Intended user of the subdirectory:

<b>I</b>	= Toshiba applications
<b>S</b>	= Operating system
<b>K</b>	= Store Systems Regression Tester
<b>U</b>	= User

- zzz** = Contents of the subdirectory:
- PGM** = Active programs and associated data
  - DT1** = Data files
  - DT4** = Data files
  - MNT** = Maintenance modules
  - BUL** = Backup level of maintenance modules

13. The following subdirectories do not follow the operating system naming conventions:

**ADX\_APGM**

User subdirectory that allows application data files to be system-mirrored files

**ADX\_BSX**

Subdirectory for .BSX symbol files that are not currently being used

**ADX\_IOSS**

Operating system print spooler data files and control blocks

Use these rules for file names and extensions.

Part of determining how to name a file is knowing the intended use of the file and the subdirectory containing the file. Generally, files are either programs or data. Some data files are really a logical extension of programs such as files that contain messages, panel definitions, or personalization values. Other data files contain data that is referred to or modified as a result of the sales activity.

Table 1 on page 18 describes the naming rules for the various types of files and subdirectories.

*Table 1. Naming files on the Operating System*

Subdirectory	File name	Extension
ADX_IPGM ADX_IMNT ADX_IBUL	<ul style="list-style-type: none"> <li>Must be 8 characters.</li> <li>First 3 characters must be the same as defined in configuration and cannot be ADX.</li> <li>Last character must be D, F, L, O, S, V, or W. See Table 2 on page 19.</li> <li>Any of the valid name characters, but do not use special characters in the 5th character position.</li> </ul>	Must be 3 characters and must be determined according to the file use table based on the last character of the file name (see Table 2 on page 19).
ADX_IDT1	<ul style="list-style-type: none"> <li>Must be 8 characters.</li> <li>First 3 characters must be the same as the files in ADX_IPGM.</li> <li>Other 5 characters can be any of the valid name characters.</li> </ul>	<ul style="list-style-type: none"> <li>Should be DAT.</li> <li>Exception might be files that are only referred to by the application and HCP using logical file name (LFN).</li> </ul>
ADX_IDT4	<ul style="list-style-type: none"> <li>Must be 8 characters.</li> <li>First 3 characters must be the same as the files in ADX_IPGM.</li> <li>Other 5 characters can be any of the valid name characters.</li> </ul>	<ul style="list-style-type: none"> <li>Should be DAT.</li> <li>Exception might be files that are only referred to by the application and HCP using LFN.</li> </ul>
ADX_SPGM ADX_SMNT ADX_SBUL ADX_SDT1	These subdirectories are for use only by the 4690 OS and must not contain any application files.	
ADX_UPGM ADX_UMNT ADX_UBUL See <i>note</i> below.	Must be 4 characters.	Must be 3 characters and must be determined according to the file use table (see Table 2 on page 19). The last character of the file name is not used to determine the extension for these subdirectories.



Table 1. Naming files on the Operating System (continued)

Subdirectory	File name	Extension
ADX_UDT1 See <i>note</i> below.	Must be 4 characters.	Must be 3 characters according to what type of translation should be done by HCP when exchanging this file with the host: <b>ASC</b> = ASCII to EBCDIC <b>BIN</b> = No translation <b>XLT</b> = User translation
JAVA JAVA\BIN JAVA\LIB	These subdirectories are used to provide Java support.	

**Note:** To be exchanged with the host processor through HCP, names in the ADX\_UPGM, ADX\_UMNT, ADX\_UBUL, and ADX\_UDT1 subdirectories must follow these guidelines. If the files in these subdirectories are used only for local use, such as for program development, follow the general file naming rules. When local files need to be exchanged with the host, you can rename them according to the renaming rules for these subdirectories.

Table 2. File Use Table

File name character	Extension	Intended file use
A	A86	Assembler source code
B	BAS	4690 source code
C	C	C source code
D	DAT	Product control files
E		Reserved; do not use
F	DAT	Messages, input sequence tables, and similar files
G		Reserved; do not use
H	H	C source code include files
I		Reserved; do not use
J	J86	4690 source code include files
K	L86	Library files
L	286	4690-executable file
M	MAP	Linkage editor map file
MF	DAT	Data files (messages)
N	INP	Linkage editor input file
O	OBJ	Relocatable object file
P	LST	Language translator listing file
Q	LIS	4690 interlisting file
R	LIN	4690 line number file
S	DAT	Display Manager screen files
T	SYM	Linkage editor symbol table file
U	BSX	Symbols file
V	OVR	4690-executable overlay file
W	SRL	Shareable runtime libraries
X	XRF	Cross-reference file
Y	SYS	4690 non-relocatable file

Table 2. File Use Table (continued)

File name character	Extension	Intended file use
Z	BAT	Batch files

For example, the LOAD ALL TERMINALS function uses these files:

**ADXCS20L.286**

The executable module or program

**ADXCS3AS.DAT**

The Display Manager panels for running the utility interactively

**ADXCS3MF.DAT**

The message file used by the utility

## Rules for naming subdirectories and files on VFS drives

The general rules for naming subdirectories and files on VFS systems are:

1. File names can be up to 256 characters long.
2. Path lengths can be up to 260 characters including the NULL terminator.
3. Do not start a subdirectory or file name with ADX. The prefix ADX is reserved for the operating system files.
4. The maximum directory depth is 60 levels including the root directory.
5. Avoid using a name for a file that matches a name of an operating system command or logical name.
6. The following characters are not allowed in subdirectories or file names:  
 ? \* : < > " / \ |  
 Characters below decimal 32  
 The decimal 127 character  
 The decimal 255 character
7. Should not use the hyphen (-) character to avoid confusion with the command line parameters.
8. A period (.) is treated just like any other character in the name. It does not indicate a file extension.

## Using logical names

A *logical name* is a way to abbreviate a file name. You can use logical names to represent either the entire file name (LFN type) or the drive and subdirectory path of the file name (LDSN type). You can use LFN, which represents the logical file name as the entire file name. You can use LDSN, which represents the logical drive and subdirectory name, as the drive and subdirectory path part of the file name. Table 3 on page 20 illustrates the use of these terms:

Table 3. LDSN and LFN terms

Type	General form for using this type	Example of a name using this type
LDSN	LDSN:filename.extension	ADX_IDT1:EALABCDE.DAT
LFN	LFN	C:\ADX_IDT1\EALPQRST

The operating system provides LDSN forms of logical names for all of the supported subdirectories. Table 4 on page 20 lists the LDSNs.

Table 4. System-provided LDSN format

LDSN	for DRIVE: \SUBDIRECTORY\
ADX_SPGM:	C:\ADX_SPGM\
ADX_SMNT:	C:\ADX_SMNT\
ADX_SBUL:	C:\ADX_SBUL\

Table 4. System-provided LDSN format (continued)

LDSN	for DRIVE: \SUBDIRECTORY\
ADX_SDT1:	C:\ADX_SDT1\
ADX_IPGM:	C:\ADX_IPGM\
ADX_IMNT:	C:\ADX_IMNT\
ADX_IBUL:	C:\ADX_IBUL\
ADX_IDT1:	C:\ADX_IDT1\
ADX_IDT4:	C:\ADX_IDT4\
ADX_UPGM:	C:\ADX_UPGM\
ADX_UMNT:	C:\ADX_UMNT\
ADX_UBUL:	C:\ADX_UBUL\
ADX_UDT1:	C:\ADX_UDT1\

## Defining logical file names

You can create, display, or modify logical file names during the configuration process. Configuration panels enable you to define three types of logical file names:

- System files
- Application files
- User files

### Notes:

1. You must not use the same name for any type of communications line or SNA link that you use for any system, application, or user logical file name.
2. 4690 logical file and device names can contain Virtual File Names (VFN), but they are limited to 127 characters. For more information on virtual file names, see “Using long file names on VFS systems” on page 28.

## System logical file names

System logical file names are reserved names used for operating system files and subdirectories. The ADXDAXyF.DAT file (where xy is the store controller ID) in the ADX\_SPGM subdirectory contains the system logical file names.

Using the configuration panels, you can change the disk drive name defined in the LFN forms of these system files. You cannot change the LDSN forms of the system logical names and you cannot add new system logical names.

## Application logical file names

A Toshiba licensed program or another program defines application logical file names. The ADXDCxyF.DAT file (where xy is the store controller ID) in the ADX\_SPGM subdirectory contains the application logical file name.

Use the configuration panels to change the disk drive name specified in the LFN form of the application logical names. You cannot change the LDSN forms of the application logical names and you cannot add new application logical names.

## Defining user logical file names

You can completely define user logical file names through configuration. You can define these file names as any allowed logical names that are not reserved by the operating system or a Toshiba licensed program. You can define either LFN or LDSN types of user logical names. The ADXDExyF.DAT file (where xy is the store controller ID) in the ADX\_SPGM subdirectory contains the user logical file names.

You can define logical names whenever you choose, but the changes do not become effective until you activate them. To activate logical names, use the Supplemental Diskettes or the Supplemental Option using the CD-ROM.

To have more displayable characters available on your store controller applications, define the logical name *disp4680* to any value. Setting a value causes the operating system to use the 4680 Controller Video Display Character set. See the *4680 BASIC: Language Reference* for a description of the 4680 and 4690 Controller Video Display Character Sets.

---

## Logical file names on a LAN (MCF Network) system

Logical file names make accessing files on other store controllers on the LAN (MCF Network) easier by letting you abbreviate lengthy file names. The system expands your abbreviated file name to the full file name when accessing the file.

For example, MYITEMS is a logical name used by your sales application for an item record file. The system expands this name to its full file name, ADXLXAAN::C:\ADX\_IDT1\MYITEMS.DAT, when another node references MYITEMS. (See the *4690 OS: User's Guide* for a discussion of nodes and node IDs.) In this example, the expanded name gets the version of the MYITEMS.DAT file that is on the master store controller (ADXLXAAN::), on the C drive (C:), in the ADX\_IDT1 subdirectory (\ADX\_IDT1\).

You can use a logical name to refer to a unique node, a unique subdirectory, or to a file. For example, ADX\_IPGM: is a logical name for the subdirectory C:\ADX\_IPGM\. Likewise, the logical name ADX\_IPGM:MYMOD.286 refers to a module whose full file name is C:\ADX\_IPGM\MYMOD.286.

More than one logical name can refer to the same file. For example, if you define both ABC and DEF as logical names for C:\ADX\_UPGM\YOURMOD.286, any reference to ABC or DEF translates to the same file name.

You could then define ABCD to be the logical name for the file ADXLXCCN::C:\ADX\_UPGM\YOURMOD.286, and ABCDE to be the logical name for the file ADXLXAAN::C:\ADX\_UPGM\YOURMOD.286. A reference to ABC or DEF would look for YOURMOD.286 only on the local controller. A reference to ABCD would look for the file only on the store controller node with the ID of CC. A reference to ABCDE would look for the file on the acting master store controller (ADXLXAAN::) wherever it is and whatever its ID is. If there was no acting master at that time, the system returns a file error message.

Every distributed file must have a logical name defined in the application logical names file or the user logical names file. The format of the logical name in the Logical Names File is:

For a compound file:

```
ADXLXAAN::drive\subdirectory\filename.extension
```

Logical name format for a mirrored file:

```
ADXLXACN::drive\subdirectory\filename.extension
```

**Note:** The system allows only one level of a subdirectory in a logical name expansion.

You can access prime versions of compound or mirrored files to read, write, delete, or rename files. However, you can only read an image version. You cannot access image versions of distributed files that are distributed at close. If you try to write, delete, rename or lock an image version of a distributed file, you will receive an error message.

## Logical names table

The operating system uses a single table of logical names for its files, store controller node names, subdirectories, and for most I/O devices such as printers and displays. A default logical names table is

supplied with the operating system at installation. The system adds application and user logical names from the application and the user logical names files to the system table at initial program load (IPL). (See “Using logical names” on page 20 for information on application and user logical names files.)

These application files can be unique for each store controller so the logical names table can also be unique for each store controller. The files used in the building of the logical names table are:

**ADXDA<sub>xy</sub>F.DAT**

Operating system logical names file (initially the system default table)

**ADXDC<sub>xy</sub>F.DAT**

Application logical names file

**ADXDE<sub>xy</sub>F.DAT**

User logical names file

**Note:** *xy* is the node ID.

## Building a logical names table

The operating system builds a logical names table at IPL time by combining the operating system, application, and user logical file names.

The operating system uses these steps to create the table:

1. The system starts with the default operating system logical names file.
2. The system adds any new names from the application logical names file.
3. If any of the application logical file names duplicate an operating system logical file name, the operating system logical file name is overlaid with the application logical file name.
4. The system uses the user logical names file to add new names or overlay existing names.

Because the user logical names file is the last to be scanned, it has precedence over both the application and operating system logical names files.

## Displaying the logical names table

To display the logical names table, from the SYSTEM MAIN MENU, select the Command Mode option by typing **7** and press **Enter**. In the root directory, type **DEFINE -S -N** (S for system, N for logical names table) and the system displays the logical names tables. To display the logical names table one panel at a time, you can use the MORE parameter. For example:

```
C:>define -s -n |more
```

**Changing the logical names table:** You can change application logical names using the DEFINE command. However, it is recommended that these names not be changed. See the *4690 OS: User's Guide* for a description of the DEFINE command.

You cannot change application logical names through the configuration process. You can change the disk drive identification for some of these names to provide disk space and better performance.

You can define user logical names during store controller configuration. Use these names to override application logical names if wanted.

---

## Using logical names to specify destination service access points

The operating system enables you to activate a unique destination service access point (DSAP) by using logical names to specify different address points. To use this function, you must first define a user logical name with the following format to define the destination SAP:

```
xxxxDSAP
```

where xxxx are the first four characters of the SNA link name. All link names must be at least four characters long and unique within those four characters. Also, because DSAP is a single byte, the first two characters must be zeroes. For example ABCDDDSAP defined as 0008 indicates that link ABCDxxxx will have a destination SAP of 08.

The value assigned to the user logical name is a 4-digit hex value representing the destination SAP ID. If no logical name is defined or if there are any errors, DSAP defaults to 0x00 for the first XID exchange and 0x04 for the remaining XID exchanges.

**Note:** Do not assign a value of 0000 to the logical name. Also, assign a value of 0004 to use the default SAP without sending any XIDs to the NULL SAP.

---

## Accessing distributed files in store controller and terminal applications

This section shows how to access distributed files from terminal nodes using logical names. The section uses an example to show how to define the file in the user logical names table and how to distribute the file to other nodes.

The example shows how a terminal user program can access a user-defined distributed file called MYFILE.DAT. The file is a compound file that is Distribute Per Update. The system distributes a compound file to all store controllers. The example shows how to put the prime version of the file on drive D of the master store controller in the ADX\_IDT4: directory.

In the example, the user program reads the local image version of the file, which reduces response time and improves performance. To update the file, the user program opens the prime version of the file on the master store controller.

Setting up the file on the LAN (MCF Network) and opening the file in the user program is a three-part process. This process consists of defining the file, creating it, and accessing it through the user program.

### Step 1. Define the file

Define the file MYFILE.DAT in the user logical names tables. You must define the file in the logical names file on each eligible store controller. (See the *4690 OS: Planning, Installation, and Configuration Guide* for information on worksheets and designating eligible LAN store controllers.)

When the file is defined in the user logical names tables, the table will have two entries placed in it for the MYFILE.DAT user file as follows:

```
1  $YFILE = D:\ADX_IDT4\MYFILE.DAT  ! Local version
2  MYFILE = ADXLxxN::$YFILE        ! Prime version
```

where the actual node ID of the current acting master store controller replaces xx.

The first entry gives the path to the local image version of the file for each store controller (for example, on drive D in the ADX\_IDT4 subdirectory, with the file name of MYFILE.DAT). Open this file when you want to access the local version of the distributed file. Your application can only read this version.

The second entry gives the path to the prime version of MYFILE on the master store controller (ADXLxxN:). Open this file when you want to update the prime version of the distributed file.

When the prime version of MYFILE.DAT is updated or deleted, DDA updates the local image versions on the other nodes using the LFN \$YFILE. Because each eligible store controller on the LAN has its own definition of \$YFILE, the drive and subdirectory of the local image versions could be different from the path of the local version on the master store controller.

**Note:** When updating any distributed file named MYFILE.DAT using the LDSN or complete file name formats, DDA will continue to use the LFN \$YFILE on the other nodes if LFN MYFILE is defined on the master store controller. If the LFN \$YFILE is not defined on the other nodes, DDA will assume \$YFILE is the expanded full file name and place the updates in the file C:\\$YFILE on the other store controllers.

**Attention:** Undesirable operations may be performed on the image versions if two or more files named MYFILE.DAT exist in separate directories on the master store controller.

To define the file in the logical names file, perform the following steps at the master store controller. You must repeat these steps for each eligible store controller on the LAN.

1. On the SYSTEM MAIN MENU panel, type **4** and press **Enter**.
2. When the INSTALLATION AND UPDATE AIDS panel appears, type **1** and press **Enter**.
3. On the CONFIGURATION panel, type **2** and press **Enter**. When prompted "Are you configuring for a LAN system?", type **Y**.
4. When the CONTROLLER CONFIGURATION panel appears, press **Enter**.
5. At the next configuration panel, select a store controller by moving the highlighted cursor to the appropriate node ID and press **Enter**. You must return to this panel to select an ID for each eligible store controller.
6. When the next CONTROLLER CONFIGURATION panel appears, use the **Tab** key to move the cursor next to User Logical File Names. Type **X** and press **Enter**.
7. When the USER LOGICAL FILE NAMES panel appears, type **1** and press **Enter**.
8. On the same panel in the file name window, enter the user logical name to be used to access this file. (The sample problem uses MYFILE as the logical name.) Press **Enter**.
9. On the DEFINE LOGICAL FILE NAMES panel, type in the expanded name to define the logical name for the distributed user file, and press **Enter**. The sample problem uses ADXLXAAN::D:\\ADX\_IDT4\\MYFILE.DAT.  
  
The node name ADXLXAAN:: defines this file to be on the master store controller. The D:\\ defines it on drive D and the double backslash ensures that drive D is accessed while operating in any subdirectory. The remaining portion of the node name, ADX\_IDT4\\MYFILE.DAT, is the standard subdirectory/filename.extension naming convention.
10. Press **F3** to go back through the configuration panels until you return to the CONTROLLER CONFIGURATION panel. On this panel, select another controller node ID.
11. Repeat steps 5 through 10 until you have defined the user logical name MYFILE on all store controllers on your network.
12. Press **F3** again to back out of the configuration process until you get to the CONFIGURATION panel. On this panel, type **5**, and press **Enter**.
13. On the next panel, type **2** (Controller Configurations) and press **Enter** to activate the changes just made.
14. After you have verified, distributed, and activated the changes, you must IPL the controllers. To do this, press the **SysRq** key, then type **C** to select controller functions.
15. On the next panel, type **2** (Controller functions) again and press **Enter**.
16. On the next panel, type **9** (Load Controller Storage) and press **Enter**.  
In the highlighted box that appears, type an asterisk (\*) to IPL all store controllers. Answer **Y** to the prompt about stopping background programs.  
If you do not want background programs stopped now, press **F3** to process, and wait until all background processing is finished before continuing. Before the next steps can be completed, the controllers **must** be IPLed. You can choose the time to do that.

## Step 2. Create the file

Create the file MYFILE.DAT in one of the following ways:



- Create a user-written program that creates a POS file statement. For example, if you are using a 4680 BASIC program, include a CREATE POSFILE statement. Be sure the statement includes the parameters COMPOUND and PERUPDATE to give the file the correct attributes.
- At the master store controller from the host specifying either CREATE or CLOD (Create and Load) using ADCS. Be sure you specify the correct attributes on the CREATE or CLOD statements (that is, SHARE=NO, VOL=3) to get the correct distribution attributes.
- Use another supported means instead of ADCS to send HCP commands to create the file.
- Use the operating system COPY command to copy the file from a diskette to the hard disk. You can create the version on the diskette as a distributed file and then the proper attributes are transferred during the copy process. If you copy a nondistributed version, use the Distributed File Utility (DFU) to define the file as distributed after you copy it. See the *4690 OS: User's Guide* for more information about the DFU.

The file is distributed to the proper store controllers by the DDA. See the *4690 OS: User's Guide* for more information.

### Step 3. Access the file from the user program

The user program uses the MYFILE user logical name to open the prime version of MYFILE.DAT on the master store controller for a READ or READ-WRITE.

The program uses the \$YFILE file to open the local version of the file on the store controller that is supporting the TCC Network that the terminal is on.

You can open the local file only for READ, because an application cannot update an image version of a file. If you create the file with a file attribute of distribute at close, the application cannot open \$YFILE because an application cannot read a file with the distribute at close attribute.

---

## Using file names in store controller applications

In store controller applications, the 4680 BASIC statements that use file names are OPEN, CREATE, RENAME, SIZE, and CHAIN. For all statements except the CHAIN statement, use any of the following formats:

*LFN*

*LDSN:filename.extension*

*drivename:\subdirectory\...\filename.extension*

The LFN allows you to place the referenced file on any drive without modifying the application. You should define the LFN to be a complete file name with the subdirectory being ADX\_IDT1. (There should be an ADX\_IDT1 subdirectory on each drive.) If you use the LDSN format, you should define the LDSN as a drive and subdirectory path. You can use the complete file name format for cases that are not supported by the LFN or LDSN format, such as a special testing environment.

When you are using the CHAIN statement, use a file name and extension with a drive and subdirectory, or use the LDSN for the drive and subdirectory. Whichever format you use, you must specify the drive and subdirectory. The possible formats are:

*LDSN:filename.extension*

*drivename:\subdirectory\...\filename.extension*

---

## Using file names in terminal applications

In terminal applications, the 4680 BASIC statements that use file names are OPEN, CREATE, RENAME, LOAD, SIZE, and CHAIN. Each of these statements requires that the node name R:: precede the file name.



When using the OPEN, CREATE, RENAME, or LOAD statements, use the LFN format for the file name. This enables you to place the referenced file on any disk drive without modifying the application. You should define the logical file name to be a complete file name with the subdirectory being ADX\_IDT1. (There should be an ADX\_IDT1 subdirectory on each disk drive.) An example of a file name using this format is:

R::EALABCDE

**Note:** The LFN represents all of the file name *except* the R:: node name. The terminal application can access any drive except the diskette drives for OPEN, CREATE, or RENAME. CHAIN and LOAD can access any drive.

The operating system converts logical file names to their fully qualified names that you defined in configuration.

When using the CHAIN statement, use a file name and extension without a drive or subdirectory because the operating system automatically uses the ADX\_IPGM subdirectory. An example of a file name using this format is:

R::EALPQRS.L286

---

## Using node names to access files

To access a file on another store controller, an application must use a file specification that includes the node name when opening a file. (See the *4690 OS: User's Guide* for a discussion of nodes and node IDs.) If file specification does not include the node name, the request tries to locate the file on the local store controller.

The complete file specification should follow the format:

*nodename::pathname\filespec*

Every node name has the form ADXLX<sub>xy</sub>N:: where *xy* is the store controller ID. Note that you end node names with a double colon. Use a single colon to indicate physical devices on the same controller, such as C:, D:, LPT1:, COM1:, and so on. Use the double colon to indicate different nodes on the network.

**Note:** The NetBIOS protocol is not supported for Virtual File System (VFS) and Network File System (NFS) drives. That is, ADXLXDDN::V: and ADXLXDDN::N: are not supported in commands (with *V* being the VFS drive and *N* being the NFS drive and *DD* being the node ID of the controller). The use of the NetBIOS protocol is only supported to use C: and D: drives. For example, only ADXLXDDN::C: and ADXLXDDN::D: are allowed in commands.

R:: is required in a terminal application when addressing an external (any controller) file. For example, when opening the XYZ file on the file server, the terminal application refers to this file as R::XYZ, where the XYZ logical name might be defined in the logical names table as ADXLXACN::C:\ADX\_IDT1\EAL\_XYZ.DAT. ADXLXACN is the system name for the file server.

In addition to the unique node name, some store controllers also have a *logical* node name. These store controllers are the master, alternate master, file server, and alternate file server. The logical node name provides a quick means of identifying the most important store controllers on your system. The logical node name is updated whenever the acting master or file server changes.

The following list shows how the operating system names the logical nodes on your system:

<b>Store Controller</b>	<b>Logical Node Name</b>
<b>Acting master</b>	ADXLXAAN::

**Acting alternate master**

ADXLXABN::

**Acting file server**

ADXLXACN::

**Acting alternate file server**

ADXLXADN::

You can access files using either the unique node name or the logical node name.

## Logging distribution errors

The system enters errors that are made when updating copies of distributed files in the Distribution Exception Log. See the *4690 OS: User's Guide* for information about this log.

---

## Using long file names on VFS systems

Because the Java programming language can require file names greater than eight characters in length, the operating system allows the use of long file names on any 4690 OS V2 or later system that is able to run the Java Virtual Machine. 4690 OS V2 or later uses a Virtual File System (VFS) architecture to allow the use of file names greater than eight characters in length.

**Note:** A CBASIC or 16-bit C application cannot access long file name support either on the controller or from the terminal.

The VFS drive setting must be enabled through system configuration. When you enable VFS drive settings, the operating system creates logical drives M: or N:, or both, depending upon whether you enable drives C: or D:, or both, respectively. The drive determines where the VFS directory will be located. However, the information is actually stored on drives C: and D:. Once you have enabled VFS, you can use drives M: and N: to provide long file name support locally, or you can configure NFS to access long file name support from a remote system.

The VFS directory is created by the controller IPL code during the first IPL after the VFS is configured. In the VFS directory are other directories whose names are represented by `~~ab`, where *a* and *b* are ASCII representations of hexadecimal digits. There can be up to 256 of these directories. In each of these directories there can be up to 256 files with a naming convention of `cd`, where *c* and *d* are ASCII representations of hexadecimal digits.

## VFS translation

The NFS server uses the VFS server to translate the directory and long file name (or Virtual File Name [VFN]) into an encoded form of Real File Names (RFN) that will work with the 4690 OS V2 or later file system and data distribution.

## Name server database

The operating system uses a name server database to contain all of the necessary information to translate VFNs to RFNs and to keep track of which nodes are using a VFN entry. This database is controlled by the name server. There is only one active name server in the system at a time, and it runs on the acting master controller. The name server on the alternate master controller has only read access to the database files and is designated as the backup name server.

## Restrictions when accessing long file name support

A Java application in the terminal cannot use the `R::M:filespec` method to access long file name support on the controller. The application must access long file name support by using a drive letter that has been defined in the terminal as being part of an NFS mount group. The mount group should point to the remote controller's exported F:, M:, or N: drives, which contain the files the application needs.

When accessing long file name support on a controller from either a terminal or controller you cannot use the ADXLXccN:: NetBIOS node name convention. To access long file name support on a remote controller, an application must use a drive letter that has been defined as being part of an NFS mount group. The mount group should point to the remote controller's exported F:, M:, or N: drives, which contain the files the application needs.

---

## Performing file functions

4680 BASIC provides statements to perform such file functions as creating, deleting, and using data files. These statements help you manage data files within your applications. Both terminal and store controller applications can use these statements.

You can also perform file functions using commands in Command Mode. See the *4690 OS: User's Guide* for an explanation of these commands.

**Note:** The term disk is used throughout the remainder of this section to indicate both a hard disk and a diskette.

## Creating files

You can create files using either the text editor or 4680 BASIC statements. For information on creating files with the text editor, see the *4690 OS: User's Guide*.

**Attention:** Creating a file using a 4680 BASIC statement establishes a new file on a disk. If you create a file with a name that already exists, the system erases the existing file before the new file is created.

Use the CREATE statement to create sequential, random, and direct files. You can execute this statement in both terminal and store controller applications. If the disk does not have the space required for the file, the CREATE statement fails.

Use the CREATE POSFILE KEYED statement to create keyed files. You can execute this statement only in the store controller. Once you create a keyed file, you cannot increase its size. To increase the keyed file, you must rebuild it. See "Keyed files" on page 38 for an explanation of rebuilding files.

On a LAN (MCF Network) system, use the CREATE POSFILE statement to create files. You must specify a distribution mode and a file type for files created with this statement. For information on distribution modes and file types, see the *4690 OS: User's Guide*.

## Space allocation

When you create a file, the operating system keeps track of the available disk space in the file allocation table (FAT). As your application creates, deletes, or expands files, the system updates the available space in the table. When you create a random or direct file, the system allocates only one cluster on the disk. This space is not initialized. If the space must be initialized to use the file, your application should write all of the initial records. You can increase the size of a random or direct file by writing records past the end of the created file.

When you create a keyed file, the system allocates disk space for the requested number of records; it also initializes the space to binary zeros.

When you create a sequential file, the system allocates only one cluster on the disk. The space allocated is not initialized. The size of a sequential file is increased as your application writes data to the file.

## File access rights

When you create a file, the operating system assigns the user ID, group ID, and access rights of the executing application to the file. The operating system places this information in the directory entry of the

file. See “Protecting files” on page 33 for information on user ID, group ID, and access rights. You cannot change this information with a 4680 BASIC application after the file is created. To change the access rights, use the FSET command in Command Mode.

You set up the user ID and group ID for applications executed in Command Mode in the system authorization file. Applications started as primary or secondary from the SYSTEM MAIN MENU are always assigned user ID=1 and group ID=2. See “System authorization” on page 282 for more information.

The system starts a 4680 BASIC application with access rights that allow READ, WRITE, and DELETE access for owners and requesters for group or world rights. Use the ACCESS statement to modify the access rights.

Creating a file also performs the same function as opening a file. See “Accessing files” on page 30 for a description of these functions.

## Deleting files

4680 BASIC provides a DELETE statement for deleting a file from the disk. To delete a file, the application must have the file open and have DELETE access rights for the file. To ensure that the file is deleted, the file must not be opened by another application. You can ensure that your application is the only user by opening the file as LOCKED. See “Sharing files” on page 31 for information on the LOCKED option.

## Accessing files

Your application gains access to a file on a disk by using the OPEN statement. Both the terminal and the store controller can use this statement.

The values on the OPEN statement determine how the application opens the file. You normally open a file according to the way it was created. For example, you open a direct file by specifying direct with the OPEN statement. You can, however, open files differently. To sequentially read all the records in a keyed file, you can open the file in direct mode by specifying direct with the OPEN statement and specifying a record length of 512. When an OPEN statement specifies keyed, the 4680 BASIC run-time library checks that you created the file as a keyed file.

The OPEN statement requests access for the file functions that the application will perform. This statement can request access to perform read, write, or delete functions. These access rights define the allowed operations for owners, group users, and world users. The operating system checks the file’s access rights against the requested functions when the application uses the OPEN statement. If the kind of functions requested are not allowed for your application, the OPEN statement fails. You should specify only the needed functions to avoid OPEN statement failures. For terminal applications, OPEN statements that request only the read function are more efficient than OPEN statements requesting the write function. See “Protecting files” on page 33 for more information on file access rights.

The OPEN statement also specifies the type of file sharing that the executing application is requesting. The type specified remains in effect for as long as the file is open. See “Sharing files” on page 31 for information on file sharing types. If the OPEN statement requests a file sharing value that conflicts with current file opens, the OPEN statement fails.

If your application is opening a sequential file to write, use the APPEND reserved word. APPEND causes data written to the file to be added at the end of the file. The WRITE statement can only add data to a sequential file. If you want to remove the data in a sequential file, you must delete the entire file or create the file again.

For keyed, random, and direct files, do not specify the BUFF and BUFFSIZE reserved words. For these file types, the 4680 BASIC runtime buffer size is equal to the value of the RECL reserved word. The operating system provides file record level data integrity for file writes of 512 or fewer bytes only.

For sequential files, the RECL reserved word is not valid, and the BUFF and BUFFSIZE reserved words determine the 4680 BASIC runtime buffer size. If you specify BUFFSIZE, the system ignores BUFF, and the buffer size is equal to the value specified with BUFFSIZE. If you do not specify BUFFSIZE, the buffer size is equal to the value specified for BUFF multiplied by 128 bytes.

You should select the sequential file buffer size according to your intended use of the file. When you are reading a sequential file, the 4680 BASIC runtime library requests file reads in increments of the buffer size. When you are writing to a sequential file with a WRITE# statement, the 4680 BASIC runtime library uses the buffer to request a file write. If you open the sequential file as LOCKED, the 4680 BASIC runtime library places as much data as fits into the buffer, independent of record size, before requesting a file write. If you open the sequential file as other than LOCKED, then 4680 BASIC runtime library requests a file write for each application WRITE statement.

The operating system provides file record level integrity only for file writes of 512 or fewer bytes. To prevent record fragmenting when using a LOCKED sequential file, force a file write in increments of complete records by using the TCLOSE statement.

The 4680 BASIC runtime buffer used for file I/O is allocated from your application heap except for the terminal medium model. Each file has its own buffer allocated.

Terminal applications can use a PRIORITY reserved word on the OPEN and CREATE statements. The PRIORITY reserved word causes the READ or WRITE requests for this file to have a higher priority for disk service at the store controller than a file not opened with the PRIORITY reserved word. Use this function only for files that are accessed in the critical response time paths in your terminal application.

## Ending access to files

Use the CLOSE statement in the store controller and the terminal when your application has no further use for a file.

Because the store controller and terminal can lose communications with each other, the operating system provides a periodic check of the terminals. A terminal can stop communicating with the store controller when a TCC Network is broken or disconnected, when the terminal is powered off, or when you have defective terminal hardware or software. Periodically, the operating system in the store controller exchanges a message with the operating system in the terminal to determine that each terminal with open files is still communicating. If a terminal does not respond, the operating system in the store controller executes a close function for the files that are opened by that terminal. The close function also releases any outstanding record locks for that terminal. If the terminal returns to operation without having to IPL, the operating system in the terminal reopens the files for that terminal.

## Sharing files

The operating system and 4680 BASIC provide facilities for specifying and managing the way several applications share a disk file. Your application can request READ and WRITE access to a file. In addition, when your application accesses a file it can request that the system restrict the READ and WRITE access for other applications.

File security attributes control whether an application can access a specified file. This access is based on the access rights of the specified file and the type of access requested by the application. See "Protecting files" on page 33 for information on file security. File sharing controls how many applications access one file.

A terminal application cannot access a keyed file in direct mode if another terminal is currently accessing the file as a keyed file. The terminal application trying to access the file in direct mode receives errors from the store controller file system as a result. The terminal application accessing the file in keyed mode must close the file before access to the file in direct mode functions correctly.

On the OPEN and CREATE statements, you can specify how you want your application to share a file. You can use three different values on these statements to specify the three types of sharing. The sharing type is in effect as long as your application has the file open.

The following list describes the sharing values and their meaning:

#### **Option Description**

##### **LOCKED**

Requests access to this file as the only application to use this file. The request is not allowed if another application has this file open. If the request is allowed, no other application can open this file for any kind of access.

##### **READONLY**

Requests access to this file that allows all other applications to have only READ access to this file.

The request is not allowed:

- If another application has this file open as LOCKED
- If another application has this file open as READONLY or UNLOCKED and has WRITE access

If this request is allowed, another application cannot open this file as:

- LOCKED
- READONLY and request WRITE access
- UNLOCKED and request WRITE access

##### **UNLOCKED**

Requests access to this file, which allows all other applications to have both READ and WRITE access to this file. This is the least restrictive access type and you should use this type whenever possible.

The request is not allowed:

- If another application has this file open as LOCKED
- If another application has this file open as READONLY and this request specifies WRITE access

If this request is allowed, another application cannot open this file as:

- LOCKED
- READONLY if this request specified WRITE access

When your application opens a file as LOCKED or READONLY, it does not need to provide for sharing write access with other applications.

When your application opens a file as UNLOCKED, it should lock records that it intends to modify. This serializes all the changes to a record of a random, direct, or keyed file. You can use sequential files UNLOCKED without the application use of record locking because the system appends each record write to the end of the file.

In a store controller application, you can lock and unlock records for random and direct files by using the LOCK and UNLOCK functions or the AUTOLOCK and AUTOUNLOCK reserved words on the READ and WRITE statements. The preferred way is the AUTOLOCK and AUTOUNLOCK. In a terminal application you can use only the AUTOLOCK and AUTOUNLOCK.

If your application opens the same file many times, you should include only one LOCK or READ AUTOLOCK at a time for the file.

For keyed files, you can use only the AUTOLOCK and AUTOUNLOCK in both the store controller and the terminal applications.



If you have many applications that lock a record in more than one file, use the same sequence of locking in each application. This prevents applications from deadlocking by having some records locked that another application is trying to lock.

An application is not allowed to lock more than five records for each keyed file at the same time.

The system might reject a request to lock a record because another application has already locked the record. When this occurs, the application should wait and try the record lock again; if the application has an operator interface, it should notify the operator that the record is temporarily not available.

Locking a record in a keyed file actually locks all of the records in the file sector containing that record. Therefore, you should avoid application designs that would require high-performance use of keyed file record locks by two or more applications.

## Copying files

You can use the COPY command in Command Mode to copy a file. The COPY command creates a new file with the name you specify. Because the COPY command is creating a new file, the new file has the user ID and group ID assigned to the COPY command when it began. The access rights for the new file are the same as the access rights of the original file. The user requesting the COPY command must have READ access rights to the original file. If a file with the new name already exists, the user needs DELETE access to that file so that COPY can delete that file.

## Renaming files

You can rename a file with the 4680 BASIC RENAME function or you can use the RENAME command in Command Mode. Both methods change the name of the file and leave the user ID, group ID, and access rights unchanged. The requester of the RENAME must have either WRITE or DELETE access rights.

## Protecting files

The operating system and 4680 BASIC provide facilities to limit file access to only authorized users. The facilities provide for controlling READ, WRITE, DELETE, and EXECUTE access to a file for three categories of users. The categories of user are owner, group, and world. The system bases file security functions on how the user ID and group ID of the requesting application match the user ID and group ID of the requested file.

The system assigns an application a user ID and group ID when the application starts. The system assigns the IDs for an operator interactive application in the store controller according to the system authorization file. This includes Command Mode commands. For your background application, the user ID is one and the group ID is two. For operating system background applications, the user ID and group ID are either one, one and zero, or zero. The terminal 4680 BASIC application IDs are always treated as a user ID of one and a group ID of two.

You assign a file a user ID, group ID, and access rights when you create the file. This section explains the access rights. The section “File access rights” on page 29 contains information on how you assign access rights to the file.

The system performs file security checking when a file is opened. The first part of security checking compares the user ID and group ID of the requesting application and the user ID and group ID of the requested file. Based on this comparison, the requesting application falls into one of three categories: owner, group, or world.

- For the requesting application to become an owner, the user ID of that application must be equal to the user ID of the file, and the group ID of the requesting application must be equal to the group ID of the file.

```

Group = Group --
      ----- OWNER access
User  = User  --

```

- For the requesting application to become a group requester, the group ID of the requesting application must be equal to the group ID of the file without the user ID of the requesting application being equal to the user ID of the file.

```

Group = Group --
      ----- GROUP access
User  ≠ User  --

```

- For the requesting application to become a world requester, the group ID of the requesting application must not be equal to the group ID of the file.

```

Group ≠ Group --
      ----- WORLD access
User  = User  --

```

When a requesting application has a user ID of zero and a group ID of zero, the system bypasses the file security checking.

The second part of security checking compares the type of access requested by the application with the type of access allowed for this file. A 4680 BASIC application can request READ, WRITE, and DELETE access on the OPEN statement. A 4680 BASIC application requests EXECUTE access to a file by attempting to chain to that file. The access rights for each file define whether this file can be accessed to read, write, delete, or execute for each category of user. Owner, group, and world user access rights are independent and do not have to provide diminishing levels of access. For example, the access rights of the operating system data files enable an owner to read, write, delete, and execute; a group user to read and execute; and a world user to read and execute. For a 4680 BASIC application to be allowed access to a file, all of the access types requested on the OPEN statement must be allowed for the application's requester category. If any of the access types for the file are not allowed, the OPEN request fails.

To rename a file, a requesting application must have either WRITE or DELETE access rights.

## Protecting subdirectories

Subdirectories can contain multiple files and can be protected in similar ways to individual files. You assign a user ID, group ID, and access rights to each subdirectory when you create it. You assign the subdirectory the user ID and the group ID of the creating application. The creating application can be the MKDIR command or a 4680 BASIC application.

The MKDIR command sets the access rights as READ, WRITE, DELETE, and EXECUTE for the owner, nothing for the group user, and nothing for the world user. You can change the access rights with the FSET command.

When a 4680 BASIC application uses the MKDIR command to create a subdirectory, the access rights are set according to the ACCESS statement. The EXECUTE access is always set for the owner and is set for the group user and the world user if any other access right is set for that user.

The access rights for a subdirectory have a different meaning than they do for a file:

### Access Right Description

<b>READ</b>	Allows the use of SIZE and DIR commands for files in the subdirectory
<b>WRITE</b>	Allows the use of CREATE, DELETE, and RENAME commands for files in the subdirectory
<b>EXECUTE</b>	Allows a program to open files in the subdirectory
<b>DELETE</b>	Allows the use of the FSET command in Command Mode for files in the subdirectory



The access rights and restrictions described for subdirectories do not apply to the root directory.

## Enabling and disabling file and subdirectory security

Each disk can have a disk label. The disk label contains the volume name for the disk, the user ID, the group ID of the label's creator, access rights for the label, and mode flags. The mode flags control the enabling and disabling of security for files and subdirectories for the entire disk. When you enable security, you control application access to files and subdirectories. When you disable security, all applications have full READ, WRITE, EXECUTE, and DELETE access to all files and subdirectories on the disk.

You use the DISKSET command to create a disk label. See the *4690 OS: User's Guide* for information on how to use DISKSET. You can also use DISKSET to enable and disable file security for a disk if you have WRITE or DELETE access rights to the disk label.

The installation process for the operating system creates the disk label for the hard disks with a user ID of zero, a group ID of zero, and security disabled. File security on the hard disks provides protection of the system files from applications and protection of the application files from other applications.

## Reading a file record

You read the data from a disk file record into BASIC variables with the READ#, READ# LINE, READ FORM#, and READ MATRIX statements. The READ# statement specifies the file to read by the I/O session number, the variables to assign the data to, the format for mapping the data record into the variables, and whether to lock the record from use by other applications. See the *4680 BASIC: Language Reference* for a description of the format string items used with the READ FORM# statement.

You read sequential file records with the READ#, READ# LINE, and READ MATRIX statements. The READ# LINE statement assigns all of the data in the current record to a single variable, and the READ# statement assigns the individual fields of the record to the individual variables specified on the READ# statement. You can also use the READ FORM# statement for sequential files, but it is less practical because the records in sequential files usually vary in length.

The READ MATRIX statement, which is used only for sequential files, allows one-dimensional string arrays to be read from disk efficiently. READ MATRIX reads a sequential file record and parses the record into an array.

You can read random file records with the same statements as sequential files. The READ FORM# statement is less practical for random file records because the fields of a random file can vary in length.

Keyed and direct file records can be read only with the READ FORM# statement. The format string items are necessary because there are no field or record delimiters within the data record.

The reads of a sequential file normally proceed from the first record of the file to the last record of the file. By using the PTRRTN function, you can save the relative position of a record within a sequential file so that you can use it to reread from that position in the file. You use the POINT statement to re-establish the saved relative position. You cannot issue a WRITE# statement after a POINT.

The reads of a keyed, direct, and random file are for records at any position in the file. For direct and random files, the record number determines the relative position in the file when it is multiplied by the record length. For keyed files, the key is transformed into a relative sector within the keyed file and that sector and any sectors chained to that sector are scanned for the record. See "Keyed files" on page 38 for more information about sectors.

## Writing a file record

You write the data to a disk file record from 4680 BASIC variables with the WRITE#, WRITE FORM# (described as part of WRITE#), and WRITE MATRIX statements. You can also use the PRINT USING#

statement in the store controller to write data to a disk record. The WRITE statements are the recommended statements. The WRITE statement specifies the file to write by the I/O session number, the variables to get the data from, the format to map the data from the variables into the record data, and whether to unlock the record for use by other applications. See the *4680 BASIC: Language Reference* for a description of the format string items used with the WRITE FORM#.

The operating system provides record integrity support to ensure that a record is written to the file correctly. The record integrity support is dependent on a record being no more than 512 bytes. The only exception to this is the WRITE MATRIX statement, which can write records larger than 512 bytes. Both terminal and store controller applications can use the WRITE MATRIX statement.

A store controller application writes sequential file records with the WRITE# statement. The WRITE# statement assigns the individual variables of the statement to the fields in the record. String expressions are placed into the record with a starting quotation mark and ending quotation mark followed by a comma. Numeric expressions are placed into the record after being converted to their ASCII representation and are followed by a comma. The last expression in the record is followed by a CR/LF instead of a comma. The WRITE FORM# statement can also be used in a store controller application to write a sequential record, but it is not very convenient because the application must provide all of the sequential file record delimiters.

When using a WRITE# statement to write a shared sequential file, the BUFFSIZE value in the 4680 BASIC OPEN statement should be as large as the largest record that you are going to write to that file. This value should take into consideration the quotes enclosing the fields and the two bytes for the CR/LF. Failure to do this could cause the record being written to be split if another application writes to that file at the same time.

A terminal application writes sequential file records with the WRITE MATRIX statement. The WRITE MATRIX statement assigns the string array elements to the fields in the record in the same way that a WRITE# statement assigns strings to a field.

The records written to a sequential file by a WRITE MATRIX statement might not be in the same chronological sequence in which they were requested. Any application reading a sequential file should provide for this sequence.

You can write random file records with the WRITE# statement. The variables are mapped to the fields within a data record the same as sequential files, except that a comma is placed after the last field and the records are padded to fill the record size. You can use the WRITE FORM# statement to write a random record, but it is not very convenient because the application must provide all of the random file record delimiters.

You can write keyed and direct file records only with the WRITE FORM# statement. The format string items are necessary because there are no field or record delimiters recorded with the data.

The writes to a sequential file proceed from the first record of the file to the last record of the file. By using the PTRRTN function, you can save the relative position of a record within a sequential file to use at a later time to read from that position in the file. You cannot use PTRRTN to write for files that are opened, UNLOCKED. Use the POINT statement to re-establish the saved relative position. You can also use a POINT statement to truncate a sequential file to zero size if the file is opened LOCKED or READONLY. See the description of the POINT statement.

The writes of a keyed, direct, and random file are for records at any position in the file. For direct and random files, the record number determines the relative position in the file when it is multiplied by the record length. For keyed files, the key is transformed into a relative sector within the keyed file and that sector and any sectors chained to that sector are scanned for the record. If the record cannot be found in those sectors, it is added to the file. See "Keyed files" on page 38 for more information about sectors.

## Ensuring data integrity across power line disturbances

The operating system saves the data in NVRAM. If a power line disturbance (PLD) occurs during a disk write and prevents the completion of the disk write, the system uses the saved data to complete the disk write when the power is restored.

### PLD protection for writing one record

All file type writes of 512 bytes or less and the WRITE MATRIX are protected by this feature. There are no reserved words on the 4680 BASIC statements to invoke or disable PLD support. The PLD recovery routine writes the entire record independent of whether the record spans disk sectors.

Any disk write that is queued in the store controller and has not been started prior to the PLD is lost when a PLD occurs. Because the terminal memory is battery-powered, a terminal program can make the WRITE request again after the power is restored.

### PLD protection for writing two records

You can use the HOLD reserved word on a WRITE statement in the store controller to PLD protect a pair of record writes issued by the same application. The HOLD function ensures that either both of the records are written or that both of the records are not written with respect to the occurrence of a PLD. You can use the HOLD function only for records of 512 bytes or less for random, direct, and keyed files.

You should use this function when an application needs to ensure that modifications are made to two files. For example, getting a current total from one file and updating a running total in another file requires that the current total be written to zero and that the running total be written to the sum of the two totals. You can also use the HOLD function to control the modification of more than two files by using a status file for one of the files.

When an application requests a write with a HOLD, the operating system actually queues the request in the store controller memory and informs the requesting application that the write is complete. When the same application requests a second write with a HOLD, the system saves both of the write requests and then performs both writes. If a PLD prevents the writes from completing, it completes the writes when the power is restored.

Because the writes with HOLD are queued in the store controller memory, you can use the HOLD function in concurrently executing applications that are sharing files. Before updating a record, your application must do a read with autolock for the record. The write with HOLD must AUTOUNLOCK the record. When the first write with HOLD is issued, it is queued and that means that the AUTOUNLOCK is not executed until the write is executed. Your application should follow the locking guidelines in “Sharing files” on page 31 to prevent deadlocks. It should also minimize the time that records are being locked when using the HOLD function.

WRITE HOLD provides PLD protection only if both files being written reside on the same store controller.

The WRITE HOLD function ensures that either both or neither of the disk requests occurs if a PLD occurs. However, if other types of failures prevent the disk requests from occurring, the second WRITE HOLD indicates the failure. Because the return code cannot distinguish, the failure indicated on the second WRITE HOLD might be for the first or second disk operation. You should look at the error results as appropriate for either disk request.

## Design considerations for file performance

When your applications use files, the functions they perform are synchronous operations. Your applications should begin any necessary I/O functions, which are asynchronous, before they begin file functions. For example, in the terminal, printing should be started first.

In your terminal applications, any file function that issues a WRITE statement forces the data to be written to the media.

In store controller applications, all file functions force the data to be written to the media. The one exception is a sequential file opened in LOCKED mode.

In your store controller applications, use as few file functions as possible that cause store controller files to be locked. This is recommended because as the number of terminals for each store controller increases, the number of locks on files increases. Each terminal program might have to wait for the LOCK request to be granted.

In general, your applications should minimize the number of file requests they make. For example, if you need two data fields from the same record, read the file once and save the second field until you need it.

## Disabling the write verify function for your hard disk drive

For hard disk drives that have the Self-Monitoring, Analysis and Reporting Technology (SMART) capabilities, you can disable the write verify function and improve system performance. This function was needed at one time as a reliability check for writing to hard disk drives. The write verify function causes all data written to the hard disk drive to be read back to force a cyclic redundancy check (CRC). If the data is corrupted, the operating system can rewrite the data to the hard disk drive. The reliability of hard disk drives has improved to the point where this function is optional.

If the file ADX\_SDT1:ADXNOWRV.DAT file exists on a hard disk drive that has SMART capabilities, the write verify function will be disabled. If the file ADX\_SDT1:ADXNOWRV.DAT file exists on a hard disk drive with no SMART capability, the write verify function will continue to be used to help protect the hard disk data. It is the responsibility of the user to create the file on the hard disk drive. The file does not need to contain any information.

---

## Keyed files

Two alternate hashing algorithms are provided to improve performance in large files having more than 40 000 records and a randomizing divisor greater than 6000, and where ASCII keys are required. See “Hashing Algorithms” on page 183 for more information about these algorithms.

You can improve the performance of keyed files in your system by following some basic guidelines:

- Ensure that all of your keyed files contain an adequate amount of free space.  
In keyed files of more than 1000 records, at least 25% of the space should be free. If the record length is greater than 169, then only one or two records will fit into one sector or 508 bytes. Allow up to 50% of free space for these files to reduce chaining of these files. When you create a keyed file, the Keyed File Utility enables you to check the amount of free space by giving you the packing factor. The packing factor gives you the percentage of records occupied.
- Eliminate long chains in a keyed file.  
The system checks chain lengths against the chaining threshold. When chains greater than the threshold are reached, the system logs a message in the system error log. Use the Keyed File Utility to determine and to reduce the chain lengths. Reduce chain lengths by allocating more space, changing the hashing algorithm, or changing the randomizing divisor. See “Hashing Algorithms” on page 183 for more information.
- When the system is writing or reading a record from a keyed file, it processes the record to find out to which block in the file the record belongs. The system should need to access the file only once to find the correct record. If many overflow chains are present, the system might need more than one read from the file. By ensuring that the records are evenly distributed throughout a keyed file, you can reduce the number of disk accesses needed to get to a particular record.
- Keyed file performance improves when the file is placed on the disk in 10 or fewer contiguous extensions. Use the CHKDSK command to determine the number of contiguous extensions of a file. See the *4690 OS: User's Guide* for information on the CHKDSK command.

---

## Chapter 3. Programming Terminal I/O Devices

This chapter shows how to program terminal I/O devices. It provides general information as well as specific information about each device. Each device has a section containing information on:

- Device characteristics
- How to access the device
- How to end access to the device
- Specific programming information about the device
- A code example for the device to perform a basic task, such as printing lines

4690 controls each device through a device driver. Various input and output statements are used to communicate with each device driver. This chapter uses 4680 BASIC to explain the methods for communicating with each device driver. For syntax and further information on these statements, see the *4680 BASIC: Language Reference*.

Terminal I/O devices can also be programmed using the C language. For syntax and more information see Chapter 19, “Designing Terminal Applications With C Language,” on page 397.

**Note:** The touch screen is programmable only through the Java mouse application programming interface (API).

---

### 2x20 Displays

This section describes all the displays that fall into the 2x20 (2 rows x 20 columns) category and provides guidelines for using these displays.

The following are 2x20 displays:

- Alphanumeric
- Operator
- 40-character Liquid Crystal Display (LCD)
- 40-character Vacuum Fluorescent Display II (VFD II)
- Two-sided VFD II
- APA Display

Two different 4690 drivers provide 2x20 display support for a terminal. The alphanumeric display is handled by the alphanumeric display driver and all of the other 2x20 displays are handled by the operator display driver.

### Characteristics

- Up to three of these displays can be attached to a terminal.

To support three 2x20 displays, you must configure one as ANDISPLAY, the second as ANDISPLAY2, and the third as ANDISPLAY3 in the Terminal Device Group for your terminal.

There are some restrictions when configuring these displays. See the *4690 OS: Planning, Installation, and Configuration Guide* for the valid combinations of 2x20 displays that can be configured.

- 2x20 display format (2 rows by 20 columns)
- Display character sets representing different code pages

There are some differences in the character sets for each of the 2x20 displays.

See the *4680 BASIC: Language Reference* for a description of the available display character sets.

You can customize the display character set for the alphanumeric display. See “Customizing the Alphanumeric Display Character Set” on page 41 for more information.

The other displays have eight variable characters available. See “Customizing the LCD and VFD II Character Sets” on page 41 for more information.

## Functions Your Application Performs

Your application can perform the following functions with the 2x20 displays:

- Select which display to use.
- Write characters to any character location.
- Read characters from any character location.
- Clear the display.
- Determine whether the driver handling your 2x20 display is the alphanumeric or operator display driver.

If the 2x20 display is configured as the system display in the terminal device characteristics (or terminal device group of legacy terminals):

- Your application shares the display with the I/O processor and system functions.

The I/O processor uses the display to display keyboard data that is specified as display-as-keyed and to display operator prompts. Your input sequence table specifies the starting character location and length of these display fields. Your application should be designed to coordinate its display data with your input sequence table.

- The system functions use the 2x20 display with a separate display buffer.

If the 2x20 display is not configured as the system display, then the only system use of the 2x20 display is to display progress indicators during IPL and dump.

## Accessing the Display

Use the OPEN statement to gain access to the display.

Use the CLOSE statement to end your application's use of the display. The CLOSE statement does not clear the display.

## Clearing the Display

Use the CLEARS statement to clear the display. The CLEARS statement writes a space character to each character location and sets the current character location to (1,1).

## Writing Characters to the Display

Use the WRITE statement to display characters on the display.

Before writing data to the display, you can set the current character location.

After the WRITE statement completes, the current character location is set to the next character location after the last character location that was written to.

## Current Character Location

A character location is defined as a row and column coordinate (row,column).

The current character location is the (row,column) of the next character to be written or read.

You use the LOCATE statement to change the current character location.

Valid row values are 1 and 2 (top to bottom). Valid column values are from 1 to 20 (left to right).

## Character Wrapping

If the number of characters you write is greater than the number of character locations remaining on the current row, the characters are written on the next row. If the current row is the last row of the display, then the characters are written to row 1. The characters will continue to wrap until all of the characters are written.



If the first byte of a double byte character falls on the last column of row 1 and the second byte wraps to the next row, the entire character is written to the next row. If the first byte of a double-byte character is at the end of the last row of the display, the character is truncated.

## 2x20 Display Character Sets

The display character set determines what is displayed on the display for each character written by your application. There are some differences in the character sets for each of the 2x20 displays.

See the *4680 BASIC: Language Reference* for a description of the available display character sets.

**Customizing the Alphanumeric Display Character Set:** You can customize your alphanumeric display character set using the Alphanumeric Display Character Set option in the configuration for your terminal.

Each character location in the character set contains a 5 x 12 dot matrix, which is used to form the character. You can redefine any character location except 'blank'. Most of the default characters are defined using 5 x 9 dots of the 5 x 12 matrix. You can define no more than 36 dots on the matrix for any given character. The display has a blank space between character locations and between the two rows. You should consider these blank spaces if you use multiple character locations to produce a graphic display.

See the *4690 OS: Planning, Installation, and Configuration Guide* for more information on customizing the alphanumeric display character set.

**Customizing the LCD and VFD II Character Sets:** An application can define up to eight customized characters for the LCD or VFD II. Each character consists of a dot matrix of 5 columns by 8 rows for the LCD display, or 5 columns by 7 rows for the VFD II.

The application defines each customized character by writing a special command to the display. This command must be exactly 10 bytes long. The first byte is the escape character 27 (1Bh). The second byte, which is the index byte, must be 00h through 07h. This index specifies the RAM address where the data is stored. The application inserts the index into a string of data to write the specialized character to the display.

The remaining eight bytes of the command are data bytes representing rows of the dot matrix. The first data byte is the top row, the second data byte is the second row, and so on. The last byte is unused for the VFD II, but it must be included in the command.

The last five bits of each data byte represent dots in the row. Any of these five bits that is turned on will turn on the corresponding bit in that row of the matrix (refer to Table 5 on page 41).

For example, the following command defines a customized character for a VFD II and stores it at RAM 00h:

```
01Bh 00h 06h 09h 1Ch 08h 1Ch 09h 06h 00h
```

Table 5. Defining Customized Characters

Data Bytes	Binary Representation of the Data Bytes	Resulting Matrix
06h	00000110b	. . x x .
09h	00001001b	. x . . x
1Ch	00011100b	x x x . .
08H	00001000b	. x . . .
1Ch	00011100b	x x x . .
09h	00001001b	. x . . x

Table 5. Defining Customized Characters (continued)

Data Bytes	Binary Representation of the Data Bytes	Resulting Matrix
06h	00000110b	. . x x .
00h	00000000b	. . . . .

The application can modify a matrix while the corresponding character is displayed on the 2x20 display, and the change appears immediately. This feature can be used to animate the character on the display. When the application closes the display, each customized character reverts to the default character.

The following CBASIC example defines a customized character that is stored at RAM location 3. Then the character is concatenated with a constant string that is written to the display:

```
%environ t

! Construct the command to define the third special character:
a$ = chr$(1bh) + chr$(3) + chr$(6) + chr$(9) + chr$(1ch) + \
    chr$(8) + chr$(1ch) + chr$(9) + chr$(6) + chr$(0)

! Open the 2X20 display and write the command to store the matrix in RAM:
open "ANDISPLAY:" as 1
clears 1
write #1; a$

! Create a data string containing this character:
b$ = " Here's a 'euro': " + chr$(3)

! Write it to the display:
write #1; b$

! Pause to admire...
wait ; 5000
stop
```

## Determining Display information

You use the GETLONG function to determine the following information about the 2x20 display:

- Current character location (row and column)
- Whether the driver handling your 2x20 display is the alphanumeric or the operator display
- The display type
- The code page the display has been set to use

See the *4680 BASIC: Language Reference* for more information on using the GETLONG function.

## Reading from the Display

Use the READ statement to read the characters that have been written to your application's display buffer.

Before reading from the display, you can set the current character location. This is the row and column where you want to start reading from the display. See “Current Character Location” on page 40 for more information. The length of the data variable specified on the READ FORM # statement determines the number of character locations read from the display. If the length requested exceeds the remaining character locations on a row, wrapping is done as described in “Character Wrapping” on page 40.

After the READ FORM # is complete, the current character location is set to the next character location after the last character that was read.



## Programming Hints for 2x20 Displays

An application written for a 2x20 display will run on any other display in the 2x20 family with the following considerations:

- Differences in display character sets

See the *4680 BASIC: Language Reference* for a description of each of the character sets.

## About DBCS Enabling

The device driver for the APA display supports four languages:

- Traditional Chinese
- Simplified Chinese
- Japan
- Korea

When the terminal initializes an APA display, the device driver detects the hardware and determines which language should be displayed.

The APA display now supports Guidance Indicators. Three types of functions are provided for guidance management: OFF, ON, and DIRECT. (Refer to Table 6 and Table 7 for additional explanation.)

Table 6. APA Display Guidance Indicators

Function	Description
OFF	Turn off the guidance represented by the asserted bit(s) in the request bit mask.
ON	Turn on the guidance represented by the asserted bits in the request bit mask.
DIRECT	Update the guidance with the requested states immediately.

Table 7. Examples of OFF, ON and DIRECT

Current	Request Type	Value	Result
0xFFFF01230	ON	0x11102220	0xFFFF03230
0xFFFF01230	OFF	0xEEEEDDDD	0x11100220
0xFFFF01230	DIRECT	0x11102220	0x11102220

The Character and Graphics Display has two rows of 12 indicator lights. The first row of indicator lights is above the main screen area. The second row is below the main screen area. The indicator lights have the following values:

First Row: [0x80000000] [0x40000000] . . . [0x00200000] [0x00100000]

Second Row: [0x00008000] [0x00004000] . . . [0x00000020] [0x00000010]

## Example

This example contains code for operating a 2x20 display. The program writes text to each line of the display and clears the display.

```
%ENVIRON T
```

```
! Declare a two-byte integer.  
INTEGER*2 DISPLAY
```

```
! 20-character message for line 1.
```

```

LINE.ONE$ = "ROW ONE SAMPLE      "

! Message for line two.
LINE.TWO$ = "ROW TWO SAMPLE      "
ON ERROR GOTO ERR.HNDLR

! Open the display.
DISPLAY = 1
OPEN "ANDISPLAY:" AS DISPLAY

! Clear the display.
CLEARS DISPLAY

! Write a greeting.
WRITE #DISPLAY; " 2x20 DISPLAY SAMPLE"

! Pause.
WAIT;2000

! Clear the greeting.
CLEARS DISPLAY

! Display the string.
WRITE FORM "C20";#DISPLAY; LINE.ONE$

! Pause.
WAIT;2000

! Set character location (2,1) and display the second line
LOCATE #DISPLAY ;2,1
WRITE #DISPLAY; LINE.TWO$

! Pause.
WAIT;2000

! Clear the display then write sample ending message
CLEARS DISPLAY
WRITE #DISPLAY; "END OF 2x20 SAMPLE"
CLOSE DISPLAY
STOP

ERR.HNDLR:
! Prevent recursion.
ON ERROR GOTO END.PROG
! Determine error type
! and perform appropriate
! recovery and resume steps.
END.PROG:
STOP
END

```

---

## Shopper Display

This section describes characteristics of the shopper display and provides guidelines for using this display.

### Characteristics

The shopper display has the following characteristics:

- You can attach one of these displays (SDISPLAY:) to a terminal
- 1x8 display format with six guidance lights
- Fixed display character set

## Functions Your Application Performs

Your application can perform the following functions with the shopper display:

- Write characters to the display and display guidance lights.
- Clear the display.
- Read data from the display.
- Get guidance light information from the display

Because the shopper display cannot be configured as the system display, your application has exclusive use of the shopper display. The only system use of the shopper display is to display IPL progress indicators during IPL.

## Accessing the Shopper Display

Use the OPEN statement to gain access to the shopper display device driver.

Use the CLOSE statement to end your application's use of the shopper display driver. The CLOSE statement does not clear the characters from the shopper display.

## Clearing the Shopper Display

Use the CLEARS statement to clear the display. The CLEARS statement writes a space character to each of the character locations.

## Writing Characters to the Shopper Display

The WRITE statement allows you to write to all or part of the shopper display and guidance lights. The specified number of characters are written right-adjusted to the display, padded on the left with blanks. Guidance lights data, established by the most recently issued PUTLONG statement, is simultaneously displayed.

### Shopper Display Character Set

The display character set determines what is displayed on the display for each character written by your application. The shopper display character set includes the numerals and a limited number of alphabetic and special characters.

Each character location contains seven bar-segments used to form the character. Some locations also contain a comma or decimal point segments. The bar-segment pattern for each character is defined within the shopper display electronics. You cannot modify this character set.

See the *4680 BASIC: Language Reference* for the definition of the shopper display character set.

## Setting the Guidance Lights on the Display

Use the PUTLONG statement to set up the guidance lights on the shopper display. The guidance lights are arranged in two columns of three lights each. The six low-order bits of the byte control the guidance lights. Bit 5 controls the upper-left light, bit 4 controls the middle-left light, bit 3 controls the lower-left light, bit 2 controls the upper-right light, and so on.

## Reading from the Display

Use the READ statement to read the characters that are specified in your application display buffer.

## Determining Shopper Display Status

Use the GETLONG statement to get the current status of the guidance lights from the shopper display.

## Example

This example contains code operating the shopper display.

```

%ENVIRON T
INTEGER*4 LIGHTS
DISPLAY1=1
DISPLAY2=2
ON ERROR GOTO ENDPROG
! Open video display.
OPEN "VDISPLAY:" AS DISPLAY2
CLEARS DISPLAY2
WRITE FORM "C20"; #DISPLAY2; "video is open"
WAIT;500
! Open shopper display.
OPEN "SDISPLAY:" AS DISPLAY1
WRITE FORM "C20"; #DISPLAY2; "shopper is open"
WAIT;500
! Clear shopper display.
CLEARS DISPLAY1
WAIT;500
! Set pointer data.
LIGHTS = 00000001h
PUTLONG DISPLAY1,LIGHTS
! Write data.
WRITE FORM "C6"; #DISPLAY1;"111.11"
WAIT;500
! Set pointer data.
LIGHTS = 00000002h
PUTLONG DISPLAY1,LIGHTS
! write data
WRITE FORM "C10"; #DISPLAY1;"22,222,222"
WAIT;500
! set pointer data
LIGHTS = 00000004h
PUTLONG DISPLAY1,LIGHTS
! Write data.
WRITE FORM "C6"; #DISPLAY1;"333.33"
WAIT;500
! Set pointer data.
LIGHTS = 00000008h
PUTLONG DISPLAY1,LIGHTS
! Write data.
WRITE FORM "C6"; #DISPLAY1;"444.44"
WAIT;500
! Set pointer data.
LIGHTS = 00000010h
PUTLONG DISPLAY1,LIGHTS
! Write data.
WRITE FORM "C8"; #DISPLAY1;"5,555,55"
WAIT;500
! Set pointer data.
LIGHTS = 00000020h
PUTLONG DISPLAY1,LIGHTS
! Write data.
WRITE FORM "C12"; #DISPLAY1;"66.666.66"
! Clear displays.
CLEARS DISPLAY1
CLEARS DISPLAY2
WRITE FORM "C30"; #DISPLAY2;" SHOPPER CLEARED - END TEST "
STOP
ENDPROG:
! Display error messages and perform appropriate recovery
! and resume steps.
STOP
END

```

---

## Video Display

This section describes the video display and provides guidelines for using this display.

Two different 4690 drivers provide video display support for the terminals.

When the video display is connected using a Feature A expansion card (4683 terminal only), the Feature A video driver is used. When the video display is connected to the video port or to a Video Graphics Array (VGA) adapter, the VGA video driver is used.

Although the application programming interface (API) to these drivers is the same, there are some restrictions when using the Feature A attribute with the VGA video driver. There are extensions to the API for the VGA video driver that overcome these restrictions as well as providing additional function. See “VGA Video Driver” on page 48 for more information.

Use ADXSERVE to determine whether your video display is a Feature A or VGA. See “ADXSERVE (requesting an application service)” on page 293 for more information. See the “Example” on page 53 for an example of how to use this application service.

To have more displayable characters available on your store controller applications, define the logical name *disp4680* to any value. Setting a value causes the operating system to use the 4680 Controller Video Display Character set. See the *4680 BASIC: Language Reference* for a description of the 4680 and 4690 Controller Video Display Character Sets.

## Characteristics

The video display has the following characteristics for each of the video display drivers:

### Feature A video driver

- You can attach up to two video displays

To support two displays you must configure one of the video displays as VDISPLAY and the other as VDISPLAY2 in the Terminal Device Group for your terminal.

- Programmable video display format

The video display can have one of four video display formats.

You configure the default video display format in the Terminal Device Group for your terminal.

Table 8 shows the four video display formats that are available:

*Table 8. Feature A Video Display Format Types*

Video Display Format	Number of Rows	Number of Columns	Restrictions
25 x 80	25	80	12- and 9-in. displays only <b>Note:</b> For Enhanced Mode 25x80 is the only video display format supported for the 4690 OS. 16x60, 16x80, and 12x40 are not supported.
16 x 60	16	60	12- and 9-in. displays only
12 x 40	12	40	None
6 x 20	6	20	5-in. display only

- Video character sets representing different code pages

See the *4680 BASIC: Language Reference* for a description of the available Feature A video display character sets.

- A cursor in the form of a blinking line can appear at the bottom of any character location on the video display.

- You can select any of eight attribute specifications for every character location on the display:
  - Reverse Video
  - Blink
  - Blank
  - Intensify
  - Underline
  - Red
  - Blue
  - Green

**Note:** If reverse video is selected the foreground color (the character) is black and the color and intensify bits apply to the background color. If reverse video is not selected then the background color is black, and the color and intensify bits apply to the foreground color.

See the *4680 BASIC: Language Reference* for a list of the colors available using the combinations of color bits and the intensify bit.

## VGA Video Driver

- You can attach only one video display  
You configure this video display as either VDISPLAY or VDISPLAY2 in the Terminal Device Group for your terminal (or Terminal Device Characteristics for SurePOS 700 Series terminals).

**Note:** In a controller/terminal the video display is VDISPLAY.

- Programmable video display format

The video can have one of four video display formats.

You configure the default video display format in the Terminal Device Group (or Terminal Device Characteristics for SurePOS 700 Series terminals) for your terminal.

Table 9 shows the four video display formats that are available:

Table 9. Video Display Format Types

Video Display Format	Number of Rows	Number of Columns	Restrictions
25 x 80	25	80	None
16 x 60	16	60	This is a 16 row x 60 column character window centered on a 16 row x 80 column character screen. Character positions 1-10 and 61-80 are blank and are not accessible by your application. Accessing character location (1,1) is actually character position (11,1). VGA supports only 40 or 80 columns while in character mode.
16 x 80	16	80	This video display format is available only through the API. It cannot be configured as the default video display format in the Terminal Device Group for your terminal.
12 x 40	12	40	None

- Video character sets representing different code pages

There are differences between the VGA and Feature A video character sets.

See the *4680 BASIC: Language Reference* for a description of the available video display character sets.

- A cursor in the form of a blinking line can appear at the bottom of any character location on the video display.

**Note:** See restrictions for the 16x60 video display format in Table 9 on page 48.

- You can use the Feature A attribute or with VGA video extensions you can use the VGA attribute

**Feature A Attribute:** The Feature A attribute is the same as described in “Feature A video driver” on page 47 with the following restrictions:

- No underline support.  
The underline bit is ignored.
- No background intensified colors supported.  
The intensify bit is ignored when the reverse video bit is set to 1.

**VGA attribute:** If using the VGA attribute, the following attribute specifications are provided:

- Red, green, blue and intensify for foreground color
- Red, green, and blue for background color
- Underline for monochrome displays
- Choice of blinking or intensify for background color

**Note:** If double-byte character set (DBCS) is enabled, only the foreground and background colors can be specified. Underlining, blinking, and intensified colors are not supported.

See the *4680 BASIC: Language Reference* for a list of the colors and underlining available using the combinations of color bits and the intensify bit.

## Functions Your Application Performs

Your application can perform the following functions with the video display:

- Control the video display format
- Determine the video display format
- Determine whether the attached video display is monochrome or color
- Write characters and attributes to any character location
- Write characters to any character location without changing the attributes
- Write attributes to any character location without changing the characters
- Read characters from any character location
- Read attributes from any character location
- Control the type of attribute for each character location (VGA extension)
- Control the attribute for each character location
- Determine the type of attribute for the current character attribute (VGA extension)
- Determine the current character attribute
- Control whether underlining is enabled on monochrome displays (VGA extension)
- Determine whether underlining is enabled on monochrome displays (VGA extension)
- Control whether blinking or background intensified colors are enabled (VGA extension)
- Determine whether blinking or background intensified colors are enabled (VGA extension)
- Control current character location
- Control whether the cursor is visible at a current character location
- Clear the video display

If the video display is configured as the system display in the terminal device group or terminal device characteristics for your terminal:

- Your application shares its video display buffer with the I/O processor.  
The I/O processor uses your applications video display buffer to display keyboard data that is specified as display-as-keyed and to display operator prompts. Your input sequence table specifies the starting

character location and length of these display fields. Your application should be designed to coordinate its video display data with your input sequence table.

- Other system functions use the video display but with a separate video display buffer.

If the video display is not configured as the system display, then the only system use of the video display is to display IPL progress indicators during IPL.

## Accessing the Video Display

Use the OPEN statement to gain access to the video display device driver. OPEN sets the current character location to row 1, column 1 (1,1).

You use the GETLONG function to determine the default video display format and whether the attached video display is color or monochrome.

Use the CLOSE statement to end your application's use of the video display driver. The CLOSE statement does not clear the video display.

## Clearing the Video Display

Use the CLEARs statement to clear the video display. The CLEARs statement writes a space character with the default attribute (Feature A 0xE0) to each character location and sets the current character location to (1,1).

## Changing the Video Display Format

The video display format sets the size and number of character locations on the video display. For example, a 12 x 40 video display format defines a video display that has 12 horizontal lines (rows) with 40 characters (columns) in each line.

Use byte MM in the PUTLONG statement to change the video display format. The Feature A video driver uses three bits and the VGA video driver uses four bits to specify a change in the video display format.

If none of the video display format bits are set in the PUTLONG statement, then the video display format does not change. If you set one of the bits to 1, the video display changes to the video display format specified by that bit.

If more than one of the video display format bits is set to 1, an error is returned to your application and the video display format does not change. If you select the current video display format, success is returned to your application, but the video display format does not change.

The video display is cleared and the current character location is set to (1,1) when the video display format is changed.

## Special Considerations for a Feature A Video Driver

If you select a video display format that is not valid for the video display type that is configured, an error is returned to your application and the video display format is not changed.

If you select a video display format that is larger than the default video display format, the driver must allocate additional memory. If memory is not available to satisfy this request, an error is returned to your application and the video display format is not changed. For this reason, it is recommended that you configure the default video display format for the largest video display format that your application uses. This prevents the need to allocate additional memory when your application changes the video display format.



## Writing to the Video Display

Use the WRITE statement to display characters and/or attributes on the video display. The default mode of writing characters to the video display automatically changes the attribute associated with each character written to the current character attribute.

Use byte CC in the PUTLONG statement to change the way characters are written to the video display. To write characters without having the attribute updated see “Writing Characters Without Changing Attributes” on page 52. To write attributes without having the character updated see “Writing Attributes Without Changing Characters” on page 52.

Before writing data to the video display, you can set the current character location and the current character attribute. After the WRITE statement is complete, the current character location is set to the first character location after the last character location written.

### Current Character Attribute

The default current character attribute is the Feature A attribute 0xE0.

You can change the current character attribute by using byte AA of the PUTLONG statement. If using video extensions, you can also use byte VV to select that byte AA contains a VGA attribute, whether blinking or background intensified colors are enabled and whether underlining is enabled for monochrome video displays.

This attribute is used by the video display driver for all default mode writes performed by your application until your application changes it with another PUTLONG statement.

If the current character attribute is a Feature A attribute, then it does not affect the attribute of characters written by the I/O processor.

The I/O processor full screen function supports only the Feature A attribute. If the current character attribute is a VGA attribute, then the following must be considered:

- If a blinking Feature A attribute is chosen in your input sequence table and your application has enabled intensified background colors, then the input sequence table attribute will display with its background color intensified rather than blinking.
- If a blue Feature A attribute is chosen in your input sequence table and your application has enabled underlining, then the input sequence table attribute will display underlined if the video display is monochrome.

### Current Character Location

A character location is defined as a row and column coordinate (row,column). Each character location contains a character and an attribute. The default current character location is (1,1).

Use the LOCATE statement to change the current character location. This is the row and column where you want to start writing characters on the video display.

The current video display format determines the valid row and column values for each character location. Valid values range from one to the maximum row and column for the current video display format.

The location of the cursor is the same as the current character location. You also use the LOCATE statement to control whether or not the cursor is visible. The current character location does not affect the location of characters written by the I/O processor.

### Character Location Wrapping

If the number of characters or attributes you write is greater than the number of character locations remaining on the current row of the video display, the characters or attributes are written on the next row.

If the current row is the last row, then the characters or attributes are written to row 1. The characters or attributes continue to wrap around until all of the characters or attributes are written.

## Video Display Character Set

The video display character set determines what is displayed on the video display for each character written by your application. Each video display driver has its own set of video character sets for the different code pages supported. Differences exist between the video display character sets used by the Feature A and VGA video drivers.

See the *4680 BASIC: Language Reference* for a description of each of the video character sets.

## Writing Characters Without Changing Attributes

Use byte CC in the PUTLONG statement to change the write mode to write characters without changing the attribute.

If you select to have characters written without changing the attribute, then the current character attribute is not used on all subsequent WRITE statements until your application changes the write mode with another PUTLONG statement. The attribute remains the same as it was before the character was written.

See the “Example” on page 53 where this mode of writing characters is used to restore a screen.

## Writing Attributes Without Changing Characters

Use byte CC in the PUTLONG statement to change the write mode to write attributes without changing the character. Also use byte VV in the PUTLONG statement to change the current character attribute type.

If you select to have attributes written without changing the character, then the data variable is interpreted as attribute data on all subsequent WRITE statements until your application changes the write mode with another PUTLONG statement.

Although the current character attribute is not used, the type of the current character attribute is used to determine whether the data variable contains Feature A or VGA attributes.

See the “Example” on page 53 where this mode of writing attributes is used to restore a screen.

## Determining Video Display Information

You use the GETLONG function to determine the following information about the video display:

- Current character attribute
- Current character attribute type (VGA extension)
- Current character location (row and column)
- Whether the cursor is visible
- Current video display format
- Whether the attached video display is monochrome or color
- Whether underlining is enabled (VGA extension)
- Whether blinking or background intensified colors are enabled (VGA extension)

## Reading from the Video Display

Use the READ FORM # statement to read the characters or attributes that have been written by your application and the I/O processor. The default mode of reading from the video display is to read the characters that have been written.

Use byte CC in the PUTLONG statement to change the read mode. To read attributes instead of characters see “Reading Attributes from the Video Display” on page 53.

Before reading from the video display, you can set the current character location. This is the row and column where you want to start reading from the video display. See “Current Character Location” on page 51 for details.

The length of the data variable specified on the READ FORM # statement determines the number of character locations read from the video display. If the length requested exceeds the remaining character locations on a row, wrapping is done as described in “Character Location Wrapping” on page 51

After the READ FORM # statement is complete, the current character location is set to the first character location after the last character location that was read.

See the “Example” on page 53 where this mode of reading characters is used to save a screen.

### Reading Attributes from the Video Display

Use byte CC in the PUTLONG statement to change the read mode to read attributes from the video display. Also use byte VV in the PUTLONG statement to change the current character attribute type.

If you select to read attributes, then all subsequent READ FORM # statements will return attributes until your application changes the read mode with another PUTLONG statement. The type of the current character attribute is used to determine whether the attributes being returned are Feature A or VGA attributes.

See the “Example” on page 53 where this mode of reading attributes is used to save a screen.

**Special Considerations When Using VGA Video Driver and Feature A Attributes:** Feature A attributes read from the video display are not guaranteed to match the attributes originally written. This is because the VGA video driver must map the Feature A attribute internally to a VGA attribute, and there are some characteristics of the Feature A attribute that are not supported. See “Feature A Attribute” on page 49 for more information.

However, rewriting these previously read attributes produces the same original result.

### Running a 2x20 Display Application on the Video Display

An application written for a 2x20 display runs on a video display with the following considerations:

- The application must change the OPEN statement to access VDISPLAY or VDISPLAY2
- Character location wrapping is different

On 2x20 displays wrapping occurs to the next row when 20 character locations are exceeded. On video displays it is dependent on the current video display format.

For example, a write to the alphanumeric display of 40 characters starting at character location (1,1) appears as two lines on the alphanumeric display. The same write to the video display appears as one line.
- Display character sets are different

See the *4680 BASIC: Language Reference* for a description of the display character sets.
- GETLONG returns different information

Only the CC (current column) and RR (current row) bytes of the GETLONG are consistent between the video and 2x20 displays.

### Example

The following example contains a BASIC application for the video display. This example writes to the video display using Feature A attributes, writes to the video display using VGA attributes, saves the screen, clears the video display, then restores the saved screen.

```
%ENVIRON T
```

```
INTEGER*4 PUTLDATA  
INTEGER*2 VDISP, VGADriver
```

```

STRING TSTATUS, CHARS, FAATTRS, VGAATTRS

! ADXSERVE subroutine
SUB ADXSERVE (RET, FUNC, PARM1, PARM2) EXTERNAL
INTEGER*4 RET
INTEGER*2 FUNC, PARM1
STRING PARM2
END SUB

! Establish an error routine.
ON ERROR GOTO ERROR1

! Open the video display.
VDISP = 2
OPEN "VDISPLAY:" AS VDISP

! Change video display format to 25x80, and grey Feature A attribute
PUTLDATA = 000800E0H
PUTLONG VDISP, PUTLDATA

! Display Feature A attribute examples title line
WRITE #VDISP; "Following are examples of Feature A attributes:"

! Set character location to (3,1), set green blinking Feature A attribute
! and display test line
LOCATE #VDISP; 3,1,OFF
PUTLDATA = 00000082H
PUTLONG VDISP, PUTLDATA
WRITE #VDISP; "Green blinking characters on black background"

! Set character location to (4,1), set red reverse Feature A attribute
! and display test line
LOCATE #VDISP; 4,1,OFF
PUTLDATA = 00000021H
PUTLONG VDISP, PUTLDATA
WRITE #VDISP; "Black characters on red background"

! Set character location to (5,1), set blue Feature A attribute
! and display test line
LOCATE #VDISP; 5,1,OFF
PUTLDATA = 00000040H
PUTLONG VDISP, PUTLDATA
WRITE #VDISP; "Blue characters on black background"

! Set character location to (6,1), set magenta Feature A attribute
! and display test line
LOCATE #VDISP; 6,1,OFF
PUTLDATA = 00000060H
PUTLONG VDISP, PUTLDATA
WRITE #VDISP; "Magenta characters on black background"

! Set character location to (7,1), set magenta intensified Feature A attribute
! and display test line
LOCATE #VDISP; 7,1,OFF
PUTLDATA = 00000068H
PUTLONG VDISP, PUTLDATA
WRITE #VDISP; "Light magenta characters on black background"

! Set character location to (8,1), set white underlined Feature A attribute
! and display test line
! Note: Will not be underlined on VGA video display
LOCATE #VDISP; 8,1,OFF
PUTLDATA = 000000F0H
PUTLONG VDISP, PUTLDATA
WRITE #VDISP; "White underlined characters on black background"

! If have a VGA video display then display test lines for some VGA attributes

```

```

VGADriver = 0
CALL ADXSERVE(RET,4,0,TSTATUS)
IF (RET = 0) AND (MID$(TSTATUS,48,1) = "1") THEN \
BEGIN

    ! Set we have a VGA video driver
    VGADriver = 1

    ! Set character location (10,1), set grey VGA attribute
    ! and display VGA attribute examples title line
    LOCATE #VDISP; 10,1,0FF
    PUTLData = 01000007H
    PUTLONG VDISP, PUTLData
    WRITE FORM "C80"; #VDISP; "Following are examples of VGA attributes:"

    ! Set character location (12,1), set green foreground and black
    ! background VGA attribute and display test line
    LOCATE #VDISP; 12,1,0FF
    PUTLData = 01000002H
    PUTLONG VDISP, PUTLData
    WRITE FORM "C80"; #VDISP; "Green characters on a black background"

    ! Set green foreground and brown
    ! background VGA attribute and display test line
    PUTLData = 01000062H
    PUTLONG VDISP, PUTLData
    WRITE FORM "C80"; #VDISP; "Green characters on a brown background"

    ! Set cyan foreground and red background
    ! VGA attribute and display test line
    PUTLData = 01000043H
    PUTLONG VDISP, PUTLData
    WRITE FORM "C80"; #VDISP; "Cyan characters on a red background"

    ! Set blue foreground, grey background,
    ! blinking VGA attribute and display test line
    PUTLData = 010000F1H
    PUTLONG VDISP, PUTLData
    WRITE FORM "C80"; #VDISP; "Blue blinking characters on grey background"

    ! Set magenta background, white foreground
    ! VGA attribute and display test line
    PUTLData = 0100005FH
    PUTLONG VDISP, PUTLData
    WRITE FORM "C80"; #VDISP; "White characters on magenta background"

    ! Set red foreground, green background
    ! VGA attribute and display test line
    PUTLData = 01000024H
    PUTLONG VDISP, PUTLData
    WRITE FORM "C80"; #VDISP; "Red characters on green background"

    ! Announce about to enable background intensified colors
    WRITE FORM "C80"; #VDISP; "About to enable background intensified colors"
    ! Wait 5 seconds
    WAIT; 5000

    ! Set character location to (18,1), set light red foreground,
    ! yellow background VGA attribute, enable intensify background colors
    ! and display test lines
    LOCATE #VDISP; 18,1,0FF
    PUTLData = 050000ECH
    PUTLONG VDISP, PUTLData
    WRITE FORM "C80"; #VDISP; "Background intensified colors enabled and blinking disabled"
    WRITE FORM "C80"; #VDISP; " for entire screen - light red characters on"
    WRITE FORM "C80"; #VDISP; " yellow background"

```

```

! Announce about to enable underlining
WRITE FORM "C80"; #VDISP; "About to enable underlining"

! Wait 5 seconds
WAIT; 5000

! Set character location to (20,1), set light blue foreground, grey background
! VGA attribute, enable intensify background colors and underlining
! and display test lines
! Note: Will be light blue characters without underlining on color display
LOCATE #VDISP; 20,1,OFF
PUTLDATA = 07000079H
PUTLONG VDISP, PUTLDATA
WRITE FORM "C80"; #VDISP; "Underlining enabled for entire screen -"
WRITE FORM "C80"; #VDISP; " white underlined characters on grey background"

! Announce about to disable underlining and background intensified colors
WRITE FORM "C80"; #VDISP; "About to disable underlining and background"
WRITE FORM "C80"; #VDISP; " intensified colors

! Wait 5 seconds
WAIT; 5000

! Set character location to (22,1), set light magenta foreground,
! blue background VGA attribute, disable intensify background colors
! and underlining and display test lines
LOCATE #VDISP; 22,1,OFF
PUTLDATA = 0100001DH
PUTLONG VDISP, PUTLDATA
WRITE FORM "C80"; #VDISP; "Background intensified colors disabled, blinking enabled,"
WRITE FORM "C80"; #VDISP; " and underlining disabled for entire screen - "
WRITE FORM "C80"; #VDISP; " light magenta characters on blue background"

ENDIF

! Set grey Feature A attribute
! and display about to clear the screen
LOCATE #VDISP; 25,1,OFF
PUTLDATA = 000000E0H
PUTLONG VDISP, PUTLDATA
WRITE FORM "C80"; #VDISP; "Now let's clear the screen"

!*****
! Save the screen for restoring later
!*****

! Read characters from video display
LOCATE #VDISP; 1,1,OFF
READ FORM "C2000"; #VDISP; CHARS

! Set read mode to read Feature A attributes, set character location
! to (1,1) and read the attributes
PUTLDATA = 000002E0H
PUTLONG VDISP, PUTLDATA
LOCATE #VDISP; 1,1,OFF
READ FORM "C720"; #VDISP; FAATTRS

! If have a VGA driver then save VGA attributes
IF (VGADRIVER = 1) THEN \
BEGIN
! Set read mode to read VGA attributes then read
PUTLDATA = 01000207H
PUTLONG VDISP, PUTLDATA
LOCATE #VDISP; 10,1,OFF
READ FORM "C1200"; #VDISP; VGAATTRS
ENDIF

```

```

! Wait 5 seconds then clear
WAIT; 5000
CLEARS VDISP

!*****
! Restore the screen
!*****

LOCATE #VDISP; 1,1,OFF
WRITE FORM "C80"; #VDISP; "Now let's restore the screen"

! Wait 3 seconds
WAIT; 3000

! Set write mode to write Feature A attributes then write them
PUTLDATA = 000008E0H
PUTLONG VDISP, PUTLDATA
LOCATE #VDISP; 1,1,OFF
WRITE FORM "C720"; #VDISP; FAATTRS

! If have a VGA driver then restore VGA attributes
IF (VGADriver = 1) THEN \
BEGIN

    ! Set write mode to write VGA attributes then write them
    PUTLDATA = 01000807H
    PUTLONG VDISP, PUTLDATA
    LOCATE #VDISP; 10,1,OFF
    WRITE FORM "C1200"; #VDISP; VGAATTRS
ENDIF

! Wait 2 seconds
WAIT; 2000

! Set write mode to write characters without updating attribute,
! set grey Feature A attribute
PUTLDATA = 000004E0H
PUTLONG VDISP, PUTLDATA
LOCATE #VDISP; 1,1,OFF
WRITE FORM "C2000"; #VDISP; CHARS

! Wait 2 seconds
WAIT; 2000

! Set character location to (25,1) and display test end line
LOCATE #VDISP; 25,1,OFF
WRITE FORM "C80"; #VDISP; "End of Video display sample program"

CLOSE VDISP
STOP

ERROR1:
! Prevent recursion.
ON ERROR GOTO END.PROG
! Determine error type and perform appropriate recovery
! and resume steps.
END.PROG:
STOP
END

```

## Using Feature A Video Driver Command Stacking to Improve Performance

You can use command stacking to further improve system performance when using video displays with the Feature A video driver.

The VGA video driver does not support command stacking. Command stacking selection on the PUTLONG statement is ignored by the VGA video driver. However, this method requires minor changes to your application programs.

Command stacking reduces the system overhead that is associated with sending the application program's WRITE, LOCATE, and PUTLONG statements. Command stacking combines or packages a number of these statements into a single video adapter command. Using this packaging or stacking technique, you can send a group of application statements to the video adapter with no more overhead than a single command.

In some cases, the video driver automatically combines the statements sent from an application program before sending the statements to the video adapter. However, you can gain better video performance by using the application program to signal the video driver when it is safe to combine application statements. The application program can use bit 0 of byte CC of the PUTLONG statement to convey this information to the driver.

An application program must set bit 0 of byte CC to 1 to allow the driver to begin stacking application statements. The driver continues to stack application statements for maximum performance until bit 0 of byte CC is reset to zero.

The video driver determines the amount of information that can be combined into one video adapter command. When bit 0 is set to 1, the video driver continues to combine application statements until the maximum adapter command size is reached. When this maximum size is reached, the video driver sends the command to the adapter and combines subsequent application statements in a second adapter command. This process continues independently of the application program. If you reset bit 0 from 1 to 0, the video driver sends all application statements in the video command buffer to the adapter. If bit 0 is not reset, the driver continues to wait for additional application program statements until the maximum adapter command size is reached.

When bit 0 is set to 1, it signals the driver to begin combining application statements. Bit 0 must be maintained as 1 on subsequent statements until the application issues a PUTLONG that resets bit 0 to zero. See the "Example Program Using Command Stacking" on page 58 for additional information.

## Restrictions Using Command Stacking

You can use command stacking only with other application WRITE statements of a similar type. For example, commands to write characters while updating attributes can be stacked only with other commands that write characters while updating attributes, write character only commands with other write character only commands, and write attribute only commands with other write attribute only commands. Switching from writing only attributes to writing only characters causes the system to flush the stack buffer. Therefore, when writing both characters and attributes, it is more efficient to set the attributes using a PUTLONG command (leaving bits 2 and 3 of byte CC 0) and then issue a WRITE command to send the character data, rather than sending separate write attribute and write character commands.

Any data the I/O processor sends to the video driver results in the command stacking buffer being cleared and the data sent to the screen. Examples of data written by the I/O processor include displaying keyboard input using the display-as-keyed feature of the input sequence table, and displaying operator prompt messages defined in the input sequence table. Because information displayed this way is considered to be immediately required by the user, the video driver forces it to the screen automatically instead of waiting for the application to flush the command stacking buffer.

## Example Program Using Command Stacking

The following example shows an application program that uses command stacking. The program displays a box on the screen with a calendar included inside the box. This program shows how to turn on



command stacking and how to leave the stacking bit turned on until a group of statements have been issued. Command stacking is optional, but might result in faster screen response when using the video display.

The source code following shows how to use the PUTLONG command to stack or combine all of the program statements required to display the calendar. Bit 1 of byte CC in the PUTLONG statement is set to 1 before any of the WRITE statements are issued. This same bit is maintained as 1 during all of the PUTLONG commands used to change the screen attributes. After all of the WRITE statements have been issued, a final PUTLONG command clears the video buffer, thus sending the data to the screen.

## Example Program Source Code

```
! ***** Command Stacking Program Example *****
!
! This program displays the December 1995 calendar on the
! 4683 video display. It utilizes the command stacking
! feature of the Feature A video driver to ensure the best screen
! response is attained.
!
! *****

%ENVIRON T

! Establish an error routine.
ON ERROR GOTO ERROR1
OPEN "VDISPLAY:" AS 2      ! Open video display driver
CLEARS 2                   ! Clear display before displaying the calendar

! Change video display format to 25x80, and set grey attribute
PUTLONG 2, 000800E0H

! Start command stacking, bit zero byte CC = 1
PUTLONG 2, 000001E0H

! ***** Form the box that surrounds the calendar
LOCATE #2; 2, 4, OFF
WRITE FORM "C31"; #2; CHR$(6)+STRING$(29, CHR$(12))+CHR$(7)
FOR II = 3 To 13
  LOCATE #2; II, 4, OFF
  WRITE FORM "C31"; #2; CHR$(25) + STRING$(29, CHR$(32)) + CHR$(24)
NEXT II

LOCATE #2; 14, 4, OFF
WRITE FORM "C31"; #2; CHR$(4) + STRING$(29, CHR$(12)) + CHR$(5)

!***** Write the Month and Year using the intensify and underscore attribute
PUTLONG 2, 000001F8H ! Change to intensify and underscore, leaving stack bit on
LOCATE #2; 4, 13, OFF
WRITE #2; "DECEMBER 1995"
!***** Write days of the week in highlighted attribute
PUTLONG 2, 000001E8H ! Change to highlighted, leaving stack bit on
LOCATE #2; 6, 6, OFF
WRITE #2; "SUN MON TUE WED THU FRI SAT"

!***** Write the days of the month using a normal attribute
PUTLONG 2, 000001E0H ! Change to normal, leaving stack bit on
LOCATE #2; 8, 28, OFF
WRITE #2; "1 2"
LOCATE #2; 9, 8, OFF
WRITE #2; "3 4 5 6 7 8 9"
LOCATE #2; 10, 7, OFF
WRITE #2; "10 11 12 13 14 15 16"
LOCATE #2; 11, 7, OFF
WRITE #2; "17 18 19 20 21 22 23"
LOCATE #2; 12, 7, OFF
```

```

WRITE #2; "24 25 26 27 28 29 30"
LOCATE #2; 13, 7, OFF
WRITE #2; "31"

!***** Make the day of the month blink for emphasis and also intensify
PUTLONG 2, 000001EAH ! Change to blinking and intensified, leaving stack bit on
LOCATE #2; 9, 11, OFF
WRITE #2; "4"

PUTLONG 2, 000000E0H      ! Flush video buffer - force data to screen

CLOSE 2
STOP

ERROR1:
! Prevent recursion.
ON ERROR GOTO END.PROG
! Determine error type and perform appropriate recovery
! and resume steps.
END.PROG:
STOP
END

```

---

## Cash Drawer Driver

This section describes the cash drawer driver and provides guidelines for using it.

### Characteristics

The cash drawer has the following characteristics:

- Each terminal supports a maximum of two cash drawers or, for all terminals that support an alarm, one cash drawer and one alarm.
- The SurePOS 700 Series (Models 72x, 74x and 78x) terminals do not support an external alarm in place of cash drawer number 2. The SurePOS 300 Series Model 350 supports only one cash drawer.
- The TCxWave 6140 Series terminals only support one cash drawer. The TCxWave 6140 Series does support the use of a USB-attached Cash Drawer.
- You can attach an external alarm using a set of relay contacts for all terminals that support external alarms. Your application program controls whether the contacts are open or closed. The cash drawers are numbered 1 and 2. You must connect the alarm in place of cash drawer number 2.
- To open a cash drawer, your application must send a pulse of a minimum duration to the cash drawer to cause it to open. If you use a Toshiba cash drawer, the pulse time is 80 ms. If you use a non-Toshiba cash drawer, you might need other pulse times. You request these pulse times during configuration. The cash drawer driver supports pulse times from 1 ms to 1048 ms; the default is 80 ms.

**Note:** Cash drawers previously labeled as IBM are considered Toshiba cash drawers.

The cash drawer on the SurePOS 300/700 Series (Models 350, 72x, 74x and 78x) terminals is driven completely through hardware ports and interrupts versus having the cash drawer appear as a device using Device Channel Support, as is done with the cash drawers on other terminals. All cash drawer activity is handled through three hardware ports. These ports can be accessed using the values in the first three A2 I/O registers located in the PCI configuration header.

Because communication between the cash drawer on the SurePOS 300/700 Series (Models 350, 72x, 74x and 78x) terminals and the 4690 OS is handled using interrupts, there is an interrupts status register. The value of the interrupts status register is located on the PCI configuration header and differentiates between interrupts received for the cash drawer and interrupts received for other serial I/O devices. 4690 OS clears this bit after determining the source of the interrupt.

| The cash drawer directly connected to the TCxWave 6140 Series terminals must be a USB-connected cash drawer. The commands available for communication with the cash drawer have not changed, but the communication pathway is through USB, instead of RS-485 or hardware Ports.

| The TCxWave 6140 Series terminals do support the attachment of the POS IO Hub, which has an RS-485 cash drawer port. Use of this RS-485 cash drawer port is supported with the TCxWave 6140 Series terminals, though only one cash drawer will still be supported at one time. In this environment, communication with the RS-485 connected cash drawer on the AnyPlace POS Hub will occur through USB, with the AnyPlace POS Hub converting for this communication.

**Note:** The SurePOS 700 Series (Models 72x, 74x and 78x) terminals do not support attaching an alarm in place of the second cash drawer.

ADXPJD0L is the name of the new driver for the cash drawer on the SurePOS 700 Series (Models 720, 740 and 780) terminals and SurePOS 300 Series (Model 350) terminals.

## Functions your Application Performs

An application program can perform the following functions with the cash drawer or alarm:

- Open a cash drawer
- Turn an alarm on or off
- Obtain cash drawer or alarm status

| **Note:** The SurePOS 700 Series (Models 72x, 74x and 78x), SurePOS 300 Series (Model 350) and TCxWave 6140 Series terminals do not support attaching an alarm in place of the second cash drawer.

## Accessing the Cash Drawer or Alarm

Use the OPEN statement to gain access to the cash drawer driver.

Use the CLOSE statement to end your application's use of the cash drawer driver.

## Controlling a Cash Drawer or Alarm

Use the WRITE FORM statement to control a cash drawer or the alarm. With a single WRITE FORM statement you can perform one of the following operations:

- Open cash drawer 1
- Open cash drawer 2
- Turn the alarm on
- Turn the alarm off

## Obtaining Cash Drawer or Alarm Status

Use the GETLONG statement to determine the status of the cash drawers and alarm. Cash drawer status provides the following information:

- Cash drawer 1 open
- Cash drawer 1 closed
- Cash drawer 1 not connected
- Cash drawer 2 open or alarm on
- Cash drawer 2 closed or alarm off
- Cash drawer 2 or alarm not connected
- No cash drawer or alarm connected

When you issue a WRITE FORM statement to open a cash drawer, the cash drawer sensor (part of the cash drawer assembly) detects the status change. Following a WRITE FORM to open a cash drawer, give the drawer time to open before requesting a cash drawer status.

## Example

This example contains code for operating a cash drawer. This program writes a message to the display, opens the cash drawer, writes another message, and then waits for the operator to close the drawer.

```
%ENVIRON T
! Declare work integers.
INTEGER*4 I4,S4
INTEGER*1 DRAWER.ONE
! Constant to open drawer 1.
DRAWER.ONE = 1
ON ERROR GOTO ERR.HNDLR
! Open the display as #2.
OPEN "ANDISPLAY:" AS 2
CLEARS 2
! Open cash drawer driver as #1.
OPEN "CDRAWER:" AS 1
WRITE #2;"CASHDRAWER DRIVER OPEN"
WAIT;2000
START.DRAWER.CONTROL:
CLEARS 2
WRITE #2; "OPEN DRAWER 1"
WAIT;2000
! Open cash drawer number 1.
WRITE FORM "I1";#1;DRAWER.ONE
DRAWER.OPEN:
! Loop while the drawer is open.
CLEARS 2
WRITE #2; "CLOSE DRAWER"
WAIT;1000
! Get the status.
I4 = GETLONG(1)
!Shift status.
S4 = SHIFT(I4,8)
! Turn off all but status.
STAT% = S4 and 000000FFH
!Return until cash drawer is closed.
IF STAT% <> 0 THEN GOTO DRAWER.OPEN \
ELSE\
! End the execution.
CLEARS 2
WRITE #2; "end of sample"
WAIT;1000
CLOSE 1
CLOSE 2
STOP
ERR.HNDLR:
! Prevent recursion.
ON ERROR GOTO END.PROG
! Determine error type
! and perform appropriate
! recovery and resume steps.
END.PROG:
STOP
END
```

---

## Coin Dispenser Driver

This section describes the coin dispenser driver and provides guidelines for using it.

### Characteristics

The coin dispenser has the following characteristics:

- Each 4683 terminal supports a maximum of two feature expansion cards. You can attach a non-Toshiba coin dispenser to one of these feature expansion cards.

- Your application should specify the amount of money to be dispensed as a four-digit integer. This number represents the number of monetary units to be dispensed (cents, for example). The definition of the term *monetary unit* depends on the country in which the coin dispenser is designed to operate.
- The operating system supports the coin dispenser and feature expansion cards only on 4683 terminals.

## Accessing the Coin Dispenser

Use the OPEN statement to gain access to the coin dispenser driver.

Use the CLOSE statement to end communication with the coin dispenser driver.

## Dispensing Coins

To dispense coins, issue a WRITE FORM statement. You can specify the number of monetary units to be dispensed as a variable or a constant in the WRITE FORM statement. If an error occurs, control passes to the ON ASYNC ERROR subprogram. Your application should allow the coin dispenser enough time between writes to complete coin dispensing.

## Example

This example program runs through one time, dispenses 47 cents, and then stops.

```
%ENVIRON T
! Declare variables.
INTEGER*4 hx%,sx%
INTEGER*2 COIN%,ANDSP%
SUB ASYNC.ERR(RFLAG,OVER)
INTEGER*2 RFLAG
STRING OVER
RFLAG = 0
OVER = ""
hx% = ERRN
ERRFX$ = ""
FOR s% = 28 TO 0 STEP -4
    sx% = SHIFT(hx%,s%)
    THE.SUM% = sx% AND 000fh
    IF THE.SUM% > 9 THEN \
        THE.SUM%=THE.SUM%+55 \
    ELSE \
        THE.SUM%=THE.SUM%+48
    A$=CHR$(THE.SUM%)
    ERRFX$ = ERRFX$ + A$
NEXT s%
CLEARS ANDSP%
WRITE #ANDSP%;"ERR=",ERR,"    ERRL=",ERRL
LOCATE #ANDSP%;2,1
WRITE #ANDSP%;"ERRF=",ERRF," ERRN=",ERRFX$
WAIT ;15000
RESUME
END SUB
ON ERROR GOTO ERRORA
! Set ON ERROR routine.
ON ASYNC ERROR CALL ASYNC.ERR
! Set ON ASYNC ERROR routine.
COIN% = 3
! Initialize session numbers.
ANDSP% = 1
OPEN "ANDISPLAY:" as ANDSP%
! Open alphanumeric display.
CLEARS ANDSP%
! Clear alphanumeric display.
WRITE #ANDSP% ; "SAMPLE COIN PROG"
! Indicate start of application.
WAIT;5000
! Wait 5 seconds.
```

```

OPEN "COIN:" AS COIN%
! Open coin dispenser.
LOCATE #ANDSP% ; 2,1
! Locate to 2nd line of display.
WRITE #ANDSP% ; "COIN DISPENSER OPEN"
WAIT;5000
AMOUNT% = 47
! Load amount to be dispensed.
CLEAR# ANDSP%
! Clear display
WRITE FORM "C8 PIC(####) C8" ;#ANDSP% ; "WRITING ",AMOUNT%," CENTS."
WRITE FORM "I4" ; #COIN% ; AMOUNT%
! Dispense coins command.
WAIT;5000
! Wait 5 seconds.
LOCATE #ANDSP% ; 2,1
WRITE #ANDSP% ; "END OF SAMPLE"
STOP
ERRORA:
! Error assembly routine.
hx% = ERRN
ERRFX$ = ""
FOR s% = 28 TO 0 STEP -4
    sx% = shift(hx%,s%)
    THE.SUM% = sx% AND 000fh
    IF THE.SUM% > 9 THEN \
        THE.SUM%=THE.SUM%+55 \
    ELSE \
        THE.SUM%=THE.SUM%+48
    A$=CHR$(THE.SUM%)
    ERRFX$ = ERRFX$ + A$
NEXT s%
CLEAR# ANDSP%
WRITE #ANDSP%;"ERR=",ERR,"  ERRL=",ERRL
LOCATE #ANDSP%;2,1
WRITE #ANDSP%;"ERRF=",ERRF%,"  ERRN=",ERRN,"  ERRFX$"
WAIT;15000
RESUME
END

```

---

## I/O Processor

This section describes the I/O processor and provides guidelines for using it.

### Characteristics

The I/O processor and the input sequence tables work together to allow the accumulation and validation of operator input from the keyboard, optical character reader (OCR) or bar code reader, magnetic wand, or scanner. This accumulation and validation can occur simultaneously with your application. When the operator enters specified amounts of input, you can have the input forwarded to your application for further processing. The input from the various devices can be formatted so that your application receives only one format regardless of the source device.

The accumulation and validation of operator input at the I/O processor level allows rapid response to operator input. For example, during point-of-sale checkout, the operator can enter item information that the I/O processor processes while your application finishes a previous item by performing a price lookup.

The I/O processor is a driver that processes operator input according to the input sequence tables. You must build the input sequence tables according to the requirements of your application. See Chapter 7, "Using the Input Sequence Table Build Utility," on page 197 for more information. You must load the input sequence tables into terminal memory before using the I/O processor.

## Input Devices on Your System

The system device driver for the I/O processor is named IOPROC: Your application accesses the following input devices via the I/O processor driver.

- Keyboard
- Magnetic wand (4683 only)
- Scanner
- Optical Character Reader (OCR)

These devices are all optional attachments for the Point of Sale Terminals. For information on attaching and configuring them, see the section on Terminal Configuration Keywords in the *4690 OS: Planning, Installation, and Configuration Guide*.

## I/O Processor Functions

The I/O processor and input sequence tables provide the following general capabilities:

- Display messages to prompt the operator for input
- Validate operator input sequences
- Edit, modulo-check, and convert data
- Display data as entered
- Map input into buffers
- Issue error tones and display messages for valid input
- Allow automatic end-of-field sequences
- Queue input to the application for processing
- Allow CLEAR key processing without application action required

If the terminal has a primary display configured that is larger than 2x20, a set of Enhanced I/O Processor functions can be used.

**Note:** If you want to develop a unique GUI to support your register application, you can use the Java I/O processor to help develop a GUI for your application. See “Java I/O Processor Functions” on page 586 for details on the Java I/O processor functionality.

These are known as the Enhanced Full-Screen I/O Processor functions as shown in the following list:

- Customization of Message Display
- Three Display Attributes Per Field
- Large Input Sequences
- Header Field Extensions
- Upper Case Display-As-Keyed
- Additional Tab Keys Definition
- WRITE Statement
- PUTLONG Statement
- Double-Quote Substitution

## Input Sequence Tables

The input sequence tables consist of three tables:

- Input state table
- Label format table
- Modulo check table

The input state table is always required to allow operator input from the keyboard, OCR, magnetic wand, or scanner. The label format table and modulo check table are optional depending on your application requirements.

The input sequence tables are used by the I/O processor to determine what operator input is allowed and how to process it.

## Definitions and Concepts

The following terms are used to describe the I/O processor and input sequence tables:

A *state* is a condition of the terminal for which there is a set of allowed inputs. A state is identified by an ID (1 through 300). The state is set by an application program, and the terminal can move from one state to another state based on operator input occurring in a state.

A *function code* is a value from 61 to 255 that delimits an input data field. For keyed input, a function code corresponds to a non-data key (such as “enter” or “total”). The function code values for keys are defined in the keyboard layout in Terminal Configuration. For example, the Enter key is assigned a function code of 95 (function codes can be reassigned during configuration). You can associate numeric or alphanumeric data with a function code. Data is concatenated to function codes in the messages passed from the I/O processor to the application. It is possible to have a function code without any accompanying data.

A function code can result from other than a physical key being pressed. For example, an option is available to pass numeric values entered without a function key to an application. This is called the automatic end-of-field option. You can specify a function code to be associated with this data.

Function codes can represent fields on labels. The data from labels is formatted into fields consisting of function codes and data. This allows the application to handle input from scanners and readers with the same processing as used for keyed input.

A *motor function code* is a function code that causes all of the accumulated data and function codes entered since the beginning of the input sequence to be queued to the application. A motor function code is also called a *motor key*.

An *input sequence* is a series of one or more operator inputs that initially starts in a state that allows an operator to begin input. The input sequence ends in a motor function code or error specified to be given to the application. The input sequence can contain up to 10 function codes with their associated data if the Enhanced Full-Screen support is not being used, or up to 127 function codes with their data if the Enhanced Full-Screen support is being used.

## Input State Table

The input state table is required to allow and process operator input from the keyboard, OCR reader, magnetic wand, or scanner. The input state table consists of information common to all states and information defined specifically for each state.

**Note:** Where you can define an action in the following descriptions, you can specify a message to be displayed and whether to remain in the current state, go to another state, or lock (disallow) input.

## Information Common to All States

The following information is common to all states:

- Data editing information to use when display-as-keyed is used for numeric data. Display-as-keyed means the numeric keys appear on the display as they are entered. You can define the thousands separator character, decimal point character, and the number of decimal digits to use to format the field on the display after the complete field is entered. Use of display-as-keyed is specified in the function code information. The position to use on the display is specified in the state information.
- A function code (key) to clear a system busy condition and a message to display when a system busy condition occurs. The system busy condition occurs when all buffers available to hold input sequences are full and queued to the application. This condition can occur when the operator enters information faster than the application can process it. If you define a function code to clear a system busy condition, the operator must press this key before any other operator input is allowed. This is useful if you want the operator to slow down the input and verify that all items entered were processed.



- Whether a double entry of the Clear key should return the terminal to the state that is defined as the beginning of the current input sequence. Refer to the information for each state for the specification of a beginning state.
- Tone type and duration to sound for errors based on device source (keyboard, OCR reader, magnetic wand, and scanner).
- Function codes that are valid in all states. You can define a function code to be valid in all states and also valid in one or more specific states. In this case, the definition of the function code in a specific state takes precedence when in that state.
- If you have an alphanumeric or ANPOS keyboard, a field passed to the application can be a function code and zero or more numeric digits, or it can be a function code and zero or more alphanumeric characters.
- Whether Enhanced Full-Screen mode is being used.
- Double-Quote Substitution character (Enhanced Full-Screen mode only). Within an input sequence, fields are enclosed in double quotes and delimited by commas. Alphanumeric and ANPOS keyboards have a double-quote character that would conflict with the double-quote used to enclose data passed from the I/O processor to the application. To prevent such a conflict a character must be defined to be used as a substitute for double-quotes within fields. This enables the application and the I/O processor to distinguish between the two classes of double-quotes. The substitution is in the input sequence that is passed to or from the application; it is not apparent to the operator. Substitution occurs for both the function code and the data within the field. The value selected for the substitution byte should be different than any function code and any keyable value.
- Additional Tab Keys Definition (Enhanced Full-Screen mode only). You may define any configurable key as an additional tab forward or tab backward key. The definition is in effect for all states. This technique also allows you to define tab keys on keyboards that don't have any tab keys, such as the 50-key keyboard.

**Information for Each State:** For each state, you can specify the following:

- State ID and name. The state ID must be a value from 1 to 300. State names are used only in the utility used to build the input sequence table. State IDs are used by the I/O processor and passed to an application.
- Valid function codes.
- Whether this state begins an input sequence. This is related to the double **Clear** key usage definition in the common information.

**Note:** When the double **Clear** key usage is selected in the common information, and the terminal operator presses **Clear** twice consecutively, the application is notified.

Notification means that the input fields entered up to this point are queued to the application.

- Allowed input devices in the state.
- Whether data should display-as-keyed and whether to edit the data (editing information is in the common information section). Display-as-keyed is defined in both state and function code definitions. If a function code has not yet been received in a state, the state definition determines the display-as-keyed properties until a function code is received.
- A message to be displayed when the terminal enters this state.
- Whether to support automatic end-of-field in this state. If supported, you can also specify the number of data keys defined to cause end-of-field and the function code to be associated with this field.
- The action to take for function codes entered but not defined in this state.
- Whether to allow data to be keyed and assigned to a function code that will be keyed following other function codes. You can do this one of two ways. You can indicate "Data not allowed" or "Data follows function code" for the function code key that will be pressed following the data. You can also indicate "Assign saved data" for the function code to which the data is to be assigned.

- Whether or not the state is a full-screen state. A non-full-screen is restricted to the top left 2x20 area of the display for display-as-keyed data and messages defined in the state table. A full-screen state does not have this restriction.
- Where to display the data on the display, if the state is not a full-screen state. If the state is a full-screen state, the locations for displaying data are specified in the function code definitions (display-as-keyed locations) and in the common information section (message locations).
- Whether data is displayed in normal or reverse order (non-full-screen states only).

**Information for Each Function Code:** For each function code allowed in a state and for each function code common to all states, you can define the following:

- Function code value (61 through 255).
- Whether this function code is a motor function code. A motor function code causes the application to be notified.
- Whether this function code is a **Clear** key.
- Whether to notify the application if an error is detected in the data associated with this function code.
- The action to take if no error is detected in the data entered.
- Whether data is allowed, required, or optional, and the action to take if this data requirement is not met.
- Whether data associated with this function code precedes or follows entry of the function code.
- Whether to assign saved data to this function code. This option is used in conjunction with the state option to allow saving of such data. Saved data is data that was entered but could not be assigned to the function code immediately preceding or following the data (if the state option is in effect). This option is useful if an input sequence contains several function codes, and a particular function code is entered separately from its associated data.
- Whether data should display-as-keyed and whether to edit the data (editing information is in the common part of the table).
- Minimum and maximum number of data digits (0 through 64) allowed and action to take if minimum and maximum requirements are not met.
- Whether to check the value of the data entered using a defined range (0 through 999999999) and the action to take if the range check is not met.
- Whether to modulo-check the data entered using the name of a modulo-check definition in the modulo check table (see “Modulo Check Table” on page 70), and the action to take if the modulo check fails.
- The relative variable position in the read buffer of the field occupied by this function code and associated data. You can allow up to 10 function codes during an input sequence, if Enhanced Full-Screen mode is not being used, or up to 127 if Enhanced Full-Screen mode is being used. The relative position defines the order of placement for a function code and its associated data in the input sequence buffer that will be queued to your application.
- Which other positions are mutually exclusive with this function code’s relative position in the buffer. If any of the positions specified are already filled during this input sequence when the function code is entered, an error is assumed. You also can define the action to take if the mutual exclusive check is not met.
- Whether the manager’s key must be turned on to enter this function code and the action to take if the manager’s key requirement is not met.
- Whether to assign previously saved data to this function code.
- For full-screen states, additional information is in each function code:
  - Location on the display
  - Display attribute byte
  - Whether data is displayed in normal or reverse order
  - The data type (numeric, alphabetic, or alphanumeric)
  - Field order

This option specifies the order in which the fields of a state are accessed. The first field is accessed when the state is entered, tab forward accesses the next field, and tab backward accesses the previous field.

– **Automatic Tab**

This option specifies whether the cursor automatically moves to the next field when a field is full of data. If this option is not selected, the cursor remains in the current field and can be moved backward for reentry of previously keyed data.

– **Upper Case Display-As-Keyed (Enhanced Full-Screen mode only)**

An additional option in the function code definitions of full-screen states allows you to flag the field as an upper-case-only field. For an input field thus defined, the I/O processor will convert any lower-case character keyed to upper-case. The data appears as upper-case on the video display and in the input sequence passed to the application.

– **Three Display Attributes per Field (Enhanced Full-Screen mode only)**

You may specify 3 attributes per input field. These attributes are the attribute in effect when the input field is current, the attribute in effect when the input field is noncurrent and nonempty, and the attribute in effect when the input field is noncurrent and empty. The reason for a second noncurrent field attribute is to enable highlighting of required fields through the use of the 'noncurrent and empty' attribute.

**Note:** When a full-screen state is entered, the display's cursor is placed in the position defined for the state's first data-entry field. The other data-entry fields of the state are accessed by the tab forward and tab backward keys.

The values of function codes assigned to the data-entry fields can be assigned according to the convenience of the application. Their purpose is to identify the field to the application; they are not necessarily a value assigned to a key position. The non-data-entry function codes of the state (such as an Enter key) provide a recommended and relatively simple means to control routing previously entered fields to the application or transfer to another state, although you can define the function codes assigned to the data-entry fields to accomplish this.

## **Label Format Table**

The label format table defines the information required to process OCR labels, magnetic ticket labels, and bar code labels (UPC, EAN, CODE39, Interleaved Two of Five [ITF]). This table is divided into information common to all label formats and information about specific label types.

### **Information Common to All Label Formats**

- Function code required for keying bar code labels.
- Function code (key) required to indicate modulus 11 when keying a label.
- Predominant label type (UPC or EAN). Specify the label type if you have both UPC and EAN labels, and the operator is allowed to bypass leading zeros when keying one of the label types. Leading zeros are not required in the predominant label type.
- Whether to convert UPC-E label format to UPC-A label format.
- Display message for errors detected while keying labels.

### **OCR Label Format**

- Format identifier
- Number of subfields in the format
- Length of each subfield (specific or variable)
- Function code to assign to each subfield

### **Magnetic Ticket Label Format**

- Format identifier

- Number of fields in this format
- Length of each field
- Function code to be assigned to each field
- Whether each field is numeric or alphanumeric
- Order in which fields are to be processed

### Bar Code Label Format

- Format identifier
- Specific label type (UPC-A, UPC-E, UPC-A+2, UPC-A+5, UPC-E+2, UPC-E+5, UPC-D1, UPC-D2, UPC-D3, UPC-D4, UPC-D5, EAN-13, EAN-8, EAN-13+2, EAN-13+5, EAN-8+2, EAN-8+5, CODE39, CODE93, CODE128, ITF, GS1-128, GS1 DataBar, GS1 DataBar Expanded)
- Number of fields in this format
- Length of each field
- Function code to assign to each field

### Modulo Check Table

The modulo check table allows you to define the characteristics of a modulo check to be performed on key-entered or scanner-entered data fields. You can define as many different modulo-check definitions as required. When you build a modulo-check definition, you assign an eight-character name to each definition. You request modulo checking of a field by referencing the name of the modulo-check definition in the input state table. The modulo check table is defined by the following information:

- Name of the definition. The name referenced by the input state table.
- Algorithm type. Toshiba, USER, or UPC/EAN price field.
- Formula. Product Add or Product Digit Add.
- Modulus. The divisor value to use in calculating the modulo.
- Weights. Default or User-defined.
- User-defined weights. The weight assigned to each digit in the field to be checked.

All fields are not required for each algorithm. You can define as many different modulo check tables as required.

### User-Defined Modulo-Check Algorithm

If you choose a user-defined modulo-check algorithm, you must choose either the Product Add or Product Digit Add formula. The Product Add formula assumes that the sum of the product of each field digit and its assigned weight, divided by the modulus factor, results in a zero remainder.

For example, if the field being modulo checked is:

1 2 3 4 5 4

where the last digit (4) is the check digit and the weights are:

4,3,6,6,2,2

and the modulus value is 5, the sum of the products becomes:

$$(1 \times 4) + (2 \times 3) + (3 \times 6) + (4 \times 6) + (5 \times 2) + (4 \times 2)$$

$$= 4 + 6 + 18 + 24 + 10 + 8$$

$$= 70$$

Then 70 divided by 5 equals 14, which has a zero remainder, and the field passes the modulo check.

The Product Digit Add is the same as the Product Add format, except that after all digit weight products have been formed, the sum of all of the digits in the products, rather than the sum of the products is calculated. As in the preceding example, the sum divided by the modulus value must result in a zero remainder to pass the modulo check.

For example, if the field to be checked is:

1 2 3 4 5 2

where the last digit (2) is the check digit and the weights are:

4,3,6,6,2,2

and the modulus value is 5, the products resulting are:

$1 \times 4 = 4$ ,  $2 \times 3 = 6$ ,  $3 \times 6 = 18$ ,  $4 \times 6 = 24$ ,  $5 \times 2 = 10$ ,  $2 \times 2 = 4$

and the sum of the resulting products are:

$= 4 + 6 + 1 + 8 + 2 + 4 + 1 + 0 + 4$

$= 30$

Then 30 divided by 5 equals 6, which has a zero remainder, and the field passes the modulo check.

### Modulo-Check Algorithm

The modulo-check algorithm uses a modulus value of 10, the Product Digit Add format, and a series of weights, beginning with 1 for the least significant digit and alternating in a 1,2,1,2,1,2,... series.

The maximum length of a field to be modulo checked is 64 digits including the check digit. The check digit must be the last digit.

### UPC/EAN Modulo-Check Algorithm

The UPC/EAN modulo-check algorithm is used for checking price fields on the bar code labels. The UPC/EAN modulo-check algorithm multiplies each digit of the field to be checked by its assigned weight. The 10's position value of each product is then added to that product if the transform sign is plus; or subtracted from that product if the transform sign is minus. If no transform sign is specified, the product is left unchanged. The unit's position value of each transformed product is added to the check value accumulation. After the check value accumulation is complete, it is divided by the modulus. If there is no remainder from the division, the check is complete with no error.

For example, if the field to be checked is:

4 2 5 0 9 9

where the first digit (4) is the check digit, and the defined weights are:

+7, -2, 2, +3, -6, +4

and the modulus value is 11, the products values are:

$4 \times 7 = 28$

$2 \times 2 = 4$

$5 \times 2 = 10$

$0 \times 3 = 0$

$9 \times 6 = 54$

$9 \times 4 = 36$

The transformed products are:

$28 + 2 = 30$

$4 - 0 = 4$

$10 = 10$

$0 + 0 = 0$

$54 - 5 = 49$

$36 + 3 = 39$

The accumulated check value is:

$$0+4+0+0+9+9 = 22$$

Then 22 divided by 11 equals 2, which has a zero remainder, and the field passes the modulo check.

## Functions Your Application Performs

An application program can perform the following functions with the I/O processor:

- Load the input sequence tables
- Wait for accumulated input
- Read accumulated operator input
- Allow or disallow operator input
- Obtain status
- Preload input sequence data via WRITE Statement (Enhanced Full-Screen only)

### Loading the Input Sequence Tables

Before you obtain access to the I/O processor, you must load the input sequence tables into memory using a LOAD statement. The input state table is required, and the label format table and modulo-check table are optional.

### Accessing the I/O Processor

Use the OPEN statement to gain access to the I/O processor driver, specifying the parameters for the maximum buffer size and the maximum number of buffers the driver can use to queue input sequences to the application. The buffer size must be large enough to contain the largest input sequence that can be generated according to the input state table. See “Receiving Data” on page 73 for the format of the input sequence data.

Use the CLOSE statement to end communication with the I/O processor. The CLOSE function locks the I/O processor and discards any data available.

### Preparing to Receive Data from the I/O Processor

The I/O processor can be locked or unlocked. In the locked state, data cannot be read. In the unlocked state, data can be read. Following an OPEN statement, the I/O processor is unlocked and in state ID number 1. You should always build your input state table so that state 1 is a valid state and is the initial state you expect the terminal to be in, following an OPEN statement issued to the I/O processor.

### Overview of Operator Input Flow

Following an OPEN statement, the I/O processor is unlocked and state 1 is set. The operator can then enter data. The following steps describe the flow of operator input through the I/O processor and to the application:

1. Operator input consists of data (optional) and function codes. Input from the keyboard consists of data keys and function keys. Label input from readers and scanners generates data and function codes separated into fields according to the label format table.
2. The I/O processor gets one of the buffers allocated for operator input sequences.
3. The I/O processor receives the data and function codes and processes them according to the definition of the state and function code in the input state table. The input sequence tables tell the I/O processor where to place the input fields in the driver buffer.
4. As you enter function codes, different state IDs can be set according to the action defined in the input state table. Each different state can alter the allowed input sequence. Each function code can also alter the allowed input sequence by defining mutually exclusive function codes.
5. When you enter a function code that is a motor function code, the system queues the buffer that holds the current input sequence information to the application.
6. The motor function code indicates if input can continue. If the action of the motor function code is to remain in the current state or set a new state, operator input remains unlocked, and input can continue. If the action is to lock the I/O processor, input cannot occur until the application unlocks the I/O processor.

7. The application can wait for operator input, read the input sequence buffer, and unlock the I/O processor as described in the following sections.

## Waiting for Received Data

You can use a WAIT statement to wait for data to be available from the I/O processor. In many cases you need to wait for data from multiple sources such as the I/O processor and the magnetic stripe reader (MSR). The WAIT statement allows you to wait for input from multiple devices. The system uses a timer to monitor the length of time that it has not received data. When valid data is available from the I/O processor, the system runs the statement following the WAIT. When you are waiting on feedback from multiple devices, use the EVENT% statement to determine if the I/O processor is the device responding to the WAIT.

If an error occurs during a WAIT (such as keyboard offline), the system gives control to your ON ERROR routine. Errors that occur in the input sequence, which you defined to be passed to your application, do not cause entry to the ON ERROR routine. The system passes these errors to you in the input sequence buffer queued to your application.

## Receiving Data

An input sequence buffer consists of a header field followed by up to 10 function code/data fields. ASCII quotation marks enclose each field, and ASCII commas separate each field. The header field is 14 bytes long if the Enhanced Full-Screen functions are not being used or 26 bytes long if the Enhanced functions are being used. The header field contains the following information:

- State ID that began the input sequence
- Current state ID
- Input device source (last contributing device if input was from multiple devices)
- Manager's key status
- Keyed label status
- Last function code received
- Error associated with the last function code
- Tab order of the last active input field (Enhanced Full-Screen mode only)
- Cursor Position (Enhanced Full-Screen mode only) — the cursor row and column position are set to 0 if the JIOP is being used.
- Number of input fields in the sequence (Enhanced Full-Screen mode only)

**Note:** If you specify the double CLEAR usage in the common information, the state ID that began the input sequence is the last state specified as one that begins an input sequence for which the application issued an UNLOCKDEV. Otherwise, the state ID that began the input sequence is the last state for which the application issued an UNLOCKDEV. Thus, if you specify double CLEAR usage, the application can issue UNLOCKDEVs (to states that do not begin an input sequence), following its initial UNLOCKDEV for the sequence, and the state ID for the UNLOCKDEV is preserved.

Each function code/data field consists of the function code followed by any data associated with that function code. See the *4680 BASIC: Language Reference* for the details of the header field.

You can issue either a READ or a READ LINE statement to receive an input sequence queued to the application.

The READ statement reads each field into the string variables named on your READ statement. You must specify 11 string variables if you are not using the Enhanced Full-Screen functions. The READ statement reads the header field into the first variable, and reads the function code/data fields into the variable corresponding to the relative position you specified for the function code in the input state table. Relative positions 1 through 10 correspond to variables 2 through 11 because the header occupies the first variable. The READ statement removes the double quotes around each field and the comma between each field before placing the fields into your string variables.



If you issue a READ LINE statement, the statement places all fields in the input sequence in your string variable. The header field is first followed by 10 function code/data fields. Enclose each field in double quotes and separate them with a comma. Your application must parse the input sequence into separate fields as required.

## Allowing and Disallowing Operator Input

The UNLOCKDEV statement unlocks the I/O processor. When the I/O processor is unlocked, it can receive the keyboard, reader, and scanner input as specified for the current state of the terminal. You can also specify a state ID to be set and a PRIORITY parameter. The I/O processor has two queues for operator input: a normal queue and a priority queue. You control which queue the operator input is placed in by setting the active queue to either the normal or priority queue. An UNLOCKDEV statement without the PRIORITY parameter sets the normal queue, and UNLOCKDEV with PRIORITY sets the priority queue. One use for the priority queue is when operator input is queued and an error is detected. You might want to receive new input from the operator before reading the queued input. In this case, you could set the priority queue and communicate with the operator. You could then return to the normal queue with an UNLOCKDEV statement to read the queued input and continue normal processing.

The LOCKDEV statement locks the I/O processor. When locked, the I/O processor rejects all keyboard, reader, and scanner input. If you specify the PURGE parameter on an LOCKDEV statement, all data queued to your application is discarded. The data is discarded from the currently active queue.

The PUTLONG statement can be used in place of UNLOCKDEV when additional function not provided by UNLOCKDEV is needed. This allows a particular field within a state to be made the current field. PUTLONG is typically used in conjunction with the WRITE statement.

## Determining Status of the I/O Processor

Use the GETLONG statement to determine the following I/O processor status:

- I/O processor locked or unlocked
- Normal or priority queue
- If PURGE was specified on the last LOCKDEV statement performed

## Preloading Input Sequence Data

An application can use the WRITE Statement (Enhanced Full-Screen mode only) to preload input sequence data. This statement is useful when an application reads an input sequence, detects an error and needs the operator to correct the error. The I/O processor places the input sequence into its internal input sequence buffer, which is where complete, validated fields are assembled into an input sequence. The I/O processor assumes that data in the buffer is valid.

Data written to the I/O processor must be of the same form as data read from the I/O processor. In particular, you must be sure to precede default data with the field's function code.

The WRITE statement also helps the application perform error recovery. For example, if the application detects something wrong with the data in one of the fields of a received input sequence, the following produces a look and feel similar to the I/O processor's recovery of data validation errors:

1. Clear the field in error from the received input sequence.
2. The terminal beeps.
3. Display an operator guidance message.
4. Write the problem field's data to the video display using current field attributes.
5. LOCATE the cursor appropriately.
6. WRITE the input sequence back to the I/O processor.
7. PUTLONG to the I/O processor (described below). The argument to PUTLONG should specify:
  - The state to unlock to.
  - That input fields are not to be initialized.
  - That the field in error is to be current.



(If the I/O processor style error correction is not wanted, steps 1, 4 and 5 are not required, and step 7 should allow initialization of input fields.)

From the operator's point of view, this is what occurs (if all steps are implemented):

1. Data is keyed into several fields on the video display, then a motor key is pressed.
2. The terminal beeps.
3. A message is displayed indicating a problem with one of the fields.
4. The field in error is current, and is displaying residual keyed data.
5. The operator enters the correct data. The first key pressed causes the residual data to be cleared from the display and the newly keyed data to appear in its place. This is the same way that the operator corrects errors detected via the I/O processor's data validation.

Note that all of the fields that were not in error are still displayed and they do not need to be re-keyed. The operator may tab to them and edit them as usual.

6. The operator presses a motor key, and the corrected input sequence is queued to the application.

**WRITE Processing:** Input sequences written to the I/O processor must be of the same form as input sequences read from the I/O processor. A write to the I/O processor always causes the I/O processor to lock and any residual data in the input sequence buffer to be cleared.

The I/O processors input sequence buffer has fixed size slots for each relative position possible with the loaded input state table. Each slot is of the size required to hold the function code and data of the longest input field defined for that relative position anywhere in the input state table. An application may write any length data up to the slot length.

If an application attempts to write field data that is too long for a slot, the data is truncated to the slot length. It is possible to write field data of a length greater than the input field length of the current state.

An application may write fewer fields to the I/O processor than an input sequence for the loaded input sequence table would dictate. In this case, the I/O processor fills its input sequence buffer with as many fields as were provided. Any remaining slots are left empty.

An application may write more fields to the I/O processor than an input sequence for the loaded input sequence table would dictate. In this case, the I/O processor fills its input sequence buffer as usual and ignores the extra data.

The application is required to provide a status field (the first field) in a write to the I/O processor. Although required, the data in this status field is not actually used by the I/O processor. Any string, even a null string, can be specified.

**Coding the WRITE Statement:** The following 4680 BASIC code fragments illustrate how to write input sequences to the I/O processor:

Example 1:

```
!
! This is how to WRITE an input sequence that
! was read via READ LINE.
! The format string "PIC(&)" is required to prevent
! BASIC from surrounding the input sequence with an
! additional pair of double-quotes.
!
STRING INPUT.SEQUENCE$
INTEGER*2 IOP%
IOP% = 20
READ#IOP;LINE INPUT.SEQUENCE$
.
.
WRITE FORM "PIC(&)" ;#IOP%;INPUT.SEQUENCE$
```

Example 2:

```
!  
! This is how to WRITE an input sequence that  
! was read via READ.  
!  
STRING STATUS$,A$,B$,C$,D$,E$,F$,G$,H$,I$,J$  
INTEGER*2 IOP%  
IOP% = 20  
READ#IOP%;STATUS$,A$,B$,C$,D$,E$,F$,G$,H$,I$,J$  
.  
.  
WRITE#IOP%;STATUS$,A$,B$,C$,D$,E$,F$,G$,H$,I$,J$
```

## Allowing Applications Direct Access to the Scanner

The 4690 OS provides a method to allow an application to access new scanner information directly from the scanner. The types of new information that are available to be queried vary by scanner device and, therefore, the definition of the types of available data is defined by the scanner manufacturers.

This direct access feature is available to applications written in C and written in Java code. The 4690 OS provides two new C-Language APIs and a new JavaPOS™ method to facilitate this access. The APIs and JavaPOS method provide a pipeline to transmit the scanner direct-access requests to the scanner device and then any information provided in response by the scanner is returned directly to the application, without any intermediate processing by the 4690 OS. The new access points are:

- ADXNSCAN CAPI
- ADXTSCAN CAPI
- Scanner.directIO() method in JavaPOS

**Note:** The APIs and JavaPOS method do provide basic error checking on the parameters passed to them from the applications. If an error is detected, the direct-access call is aborted and an appropriate negative decimal error code is sent back to the calling application. For other errors, such as a faulty scanner device status or a scanner device that ignores the command, the command request is aborted and a negative hexadecimal long value is returned to the application indicating the reason for aborting the direct-access call. To see the information on all return codes, reference CAPITHDR.H. See “Header File” on page 399 for additional information about the CAPITHDR.H file. The APIs and JavaPOS methods do not perform any error checking on the contents of the direct-access command being sent or on the data being returned by the scanner device, because those fields are defined by the scanner manufacturers and are not processed by the 4690 OS.

**ADXNSCAN:** This API allows a C-Language application to determine which scanner devices are currently attached to the terminal and the logical unit numbers of those scanners. An application opens the I/O processor, sends the DirectIO request, and then closes the I/O processor following the return from the request, if desired. The I/O processor determines which scanner devices exist and then it retrieves the unit number and device address of the scanners that are attached. The application, also, receives a return code to indicate the status of the direct-access attempt.

A maximum of two scanners can be attached at a time. Therefore, the results from this API call are returned in a long (4-byte) field. Starting from the rightmost byte, the first byte represents the device address of the first device, the second byte represents the unit number of the first device, the third byte contains the device address of the second device, and the fourth byte contains the unit number of the second device. If only one device exists, or zero devices exist, 0xFF is put in the bytes representing the nonexistent device. If any errors occur, the long field is also filled by 0xFFFFFFFF and it is accompanied by a return code indicating the error.

The prototype for the ADXNSCAN API call is as follows:

```
LONG adx_nscan( UWORD   sessionNum,  
                ULONG    *retbuffer )
```

The parameter fields are:

- `sessionNum` — The session number received by the application when it opened the I/O processor.
- `retbuffer` — Return information from the call to the ADXNSCAN API. The unit number is either 0x00 or 0x01, representing the two possible scanner devices. The device type is either 0x4A or 0x4B. A flatbed-type scanner is represented by 0x4A and a handheld-type scanner is represented by 0x4B. For instance, if `retbuffer` equals 0xFFFF004B, then this terminal has one scanner attached, it is Unit 0, and it is a handheld-type device.

**ADXTSCAN:** This API allows a C-Language application to deliver the scanner-defined set of commands to the scanner device. An application opens the I/O processor, makes a call to this API with the desired parameters and, if wanted, closes the I/O processor following the return of the information. The ADXTSCAN API delivers the direct-access call to the scanner device through the I/O processor and the scanner driver. Any information resulting from the ADXTSCAN request is placed by the scanner driver in a return buffer and that return buffer is then passed back to the calling application. The composition of any returned information is set by the scanner device manufacturers. The application always receives a return code to indicate the status of the direct-access attempt.

The prototype for the ADXTSCAN API call is as follows:

```
LONG adx_tscan( UWORD          sessionNum,
                CHAR           _far *databuff,
                USHORT          dbuffsize,
                CHAR           _far *returnbuff,
                USHORT          rbuffsize,
                USHORT          scanNumSel )
```

The parameter fields are:

- `sessionNum` — The session number received by the application when it opened the I/O processor.
- `databuff` — A pointer to the direct access command to be delivered to the scanner device. This command can be either a Direct IO command or a Longer Direct IO command.  
**Direct IO** — This command can be up to 11 bytes long. The first byte must be 0x30 to indicate to the operating system that the command is a Direct IO call. The remaining 10 bytes are used to hold the scanner-specific subcommands and data.  
**Longer Direct IO** — This command can be up to 240 bytes long. The first byte must be 0x35 to indicate to the operating system that the command is a Longer Direct IO call. The remaining 239 bytes are used to hold the scanner-specific subcommands and data. For both the Direct IO and Longer Direct IO commands, the composition of the subcommands and data must be defined by the scanner manufacturer, and it can vary by scanner type.
- `dbuffsize` — The total length of the command being delivered to the scanner device. The length can be up to 11 bytes long for the Direct IO command, and up to 240 bytes for the Longer Direct IO command.
- `returnbuff` — A pointer to the information returned from the scanner device in response to the direct access request. The `returnbuff` must be at least two bytes long because the first two bytes of the `returnbuff` hold the length of the return information, or 0, if no return information exists.

#### Notes:

1. The amount of space for returned data in the `returnbuff` is always two less than the total declared size of the `returnbuff`, because of these length bytes.
2. The length can be accessed by treating the first two bytes of the `returnbuff` as a WORD. The length given by the length bytes, however, only indicates the amount of data returned and does not include the size of the two length bytes themselves. Therefore, the buffer is actually two bytes longer than the indicated length. To access the actual data, if any is returned, always begin with the third position in the buffer.
3. The 4690 OS does not do any interpretation of the return information, but simply delivers it to the application.

If the return code from this API is 0, this indicates that all the data from the scanner device was able to fit in the `returnbuff` and the two length bytes at the beginning of the `returnbuff` then indicate the amount of data that was returned.

However, if the return code from this API is 1, this indicates that the scanner device generated more data than could be fit in the returnbuff, so only as much data as could fit in the buffer is in the returnbuff. In this case, the first two bytes of returnbuff indicate the total amount of data generated by the scanner device, from which can be determined the necessary size of the returnbuff to hold all of the data. To calculate the amount of data actually returned, in this case, subtract two (for the length bytes) from the total declared size of the returnbuff created by the application.

If the return code to the API is anything other than 0 or 1, no data is being returned to the application. To determine the reason for this, reference CAPITHDR.H for the meaning of the return code. In this case, the first two bytes of the returnbuff are set to 0 to indicate that no data was returned.

- **rbuffsize** — The length of the returnbuff that holds any information returned by the scanner device. This length must be at least two, as two length bytes always appear at the beginning of the returnbuff.
- **scanNumSel** — The unit number of the scanner device being referenced. This number is either 0 or 1, as there is a maximum of two scanner devices attached to a terminal at one time. To determine which scanner device is unit 0 and which is unit 1, refer to the Application Direct Access ADXNSCAN API Section.

**Scanner.directIO() method:** To use Application Direct Access to Scanner Devices from a Java application, an application calls a method within JavaPOS. The method name is `directIO()` and can be accessed by opening a scanner unit and making the method call on that scanner unit. This method provides an ADXTSCAN-type interface, which allows applications to query scanner devices for device-specific information. Because JavaPOS allows an application to open a scanner unit by referring to the specific type of scanner device the application would like to open, there is no need for an ADXNSCAN-type interface for Java. Any information resulting from the direct access call is placed in the ReturnBuffer object. A return code is always placed in the ReturnBuffer object, unless the parameters are invalid, in which case a `JposException` is thrown by JavaPOS.

The prototype for the `directIO()` method call is as follows:

```
public void directIO(int      command,
                    int[]    data,
                    Object retBuff) throws JposException
```

The parameter fields are:

- **command** — The type of command being sent to the scanner. This command must be a 0x81 to indicate to the operating system that the command is a direct access call.
- **data** — An array holding the direct access command to be delivered to the scanner device. This command can be either a Direct IO command or a Longer Direct IO command.

**Direct IO** — This command can be up to 11 bytes long. The first byte must be 0x30 to indicate to the operating system that the command is a Direct IO call. The remaining 10 bytes are used to hold the scanner-specific subcommands and data.

**Longer Direct IO** — This command can be up to 240 bytes long. The first byte must be 0x35 to indicate to the operating system that the command is a Longer Direct IO call. The remaining 239 bytes are used to hold the scanner-specific subcommands and data.

**Note:** For both the Direct IO and Longer Direct IO commands, the composition of the subcommands and data must be defined by the scanner manufacturer, and it can vary by scanner type.

- **retBuff** — This object is really of type ReturnBuffer Class and it holds the return code and any data that is returned by the scanner device in response to the direct access query. The following methods are provided by the ReturnBuffer Class to an application to access the returned information:

Constructor:

```
public ReturnBuffer()
```

Methods to access data from the scanner device:

```
public String getReturnBuff()
public byte[] getReturnBuffBytes()
```

Method to clear the return information between uses of the ReturnBuffer Object:

```
public boolean clearReturnBuff()
```

Method to access the return code:

```
public int getReturnCode()
```

Method to access the length of data from the scanner device:

```
public short getLength()
```

## Non-Full-Screen Example

The code shown here writes a message to the display, loads an input sequence table, waits for scanner input, checks the input, and writes it in readable form to the display.

```
! This routine requires that an input sequence table be created.
! In this example the table is loaded from the fixed disk in drive C.
! The table should have at least one function code, which is a motor
! function code.
! The relative position should be 3, 2, or 1 for this routine to
! display the input.
%ENVIRON T
! Declare integers.
INTEGER*2 DISPLAY
INTEGER*2 OPERIN
DISPLAY = 1
OPERIN = 2
ON ERROR GOTO ERR.HNDLR
! Open the display.
OPEN "ANDISPLAY:" AS DISPLAY
! Write a greeting.
CLEARS DISPLAY
WRITE #DISPLAY; "I/O PROCESSOR SAMPLE"
WAIT;2000
! Open the I/O processor and load an
! Input sequence table.
LOAD "ISTBL=R::C:UUUU@SMP.DAT, FMTTBL=
R::C:UUUU@LBL.DAT,MODTBL=R::C:UUUU@MOD.DAT"
OPEN "IOPROC:" AS OPERIN BUFFSIZE 70
! Prompting is displayed by the I/O processor
! as specified in the input sequence table.
KBWAIT:
! Wait for input.
WAIT;5000
! You would normally test
! for a timeout condition.
! Read the available input data.
READ #OPERIN; IOPDATA$,B$,C$,D$,E$,F$,G$,H$,I$,J$,K$
! Display 14 status bytes.
CLEARS DISPLAY
WAIT;2000
WRITE #DISPLAY;"IOP=", IOPDATA$
WAIT;2000
! Check byte 14 for I/O processor flagged edit condition.
! Return for reentry by user if error in data.
IF MID$(IOPDATA$,14,1) <> " " THEN GOTO KBWAIT
! Display three of the relative variables.
CLEARS DISPLAY
WRITE #DISPLAY;"B=", B$
WAIT;2000
CLEARS DISPLAY
WRITE #DISPLAY;"C=", C$
WAIT;2000
CLEARS DISPLAY
WRITE #DISPLAY;"D=", D$
WAIT;2000
CLEARS DISPLAY
WRITE #DISPLAY; "END OF SAMPLE"
```

```

CLOSE DISPLAY, OPERIN
! STOP
ERR.HNDLR:
! Prevent recursion.
ON ERROR GOTO END.PROG
! Determine error type
! and perform appropriate
! recovery and resume steps.
END.PROG:
STOP
END

```

## Additional I/O Processor Features

### Data Editing

Data input fields are considered to be one-dimensional arrays up to 80 characters long. They can start on one row and continue on the following rows, but in terms of data entry and cursor movement, only left and right movement is allowed. When the last position of a row is reached, the next position is the first position of the next row. The position that precedes the first position of a row is the last position of the previous row. For cursor movement keys, the cursor wraps from the last position of a field to the first position. For data entry keys, wraparound does not occur; the cursor remains at the last position of a field when the field is full and automatic tab is not in effect. Entry of more data results in an error tone.

- Data keys

The data keys insert characters and move the cursor to the right within the current field. The space bar is considered a data key that inserts a blank.

If the field is displayed in reverse order of entry, the cursor stays in the same position, data is inserted in that position, and previously entered data moves to the left.

- Backspace key

The Backspace key moves the cursor to the left and sets the moved-from position to null if no data follows it, or to blank if data follows it. The cursor stops at the first position of the field.

If the field is displayed in reverse order of entry, the cursor stays in the same position and the data to the left of that position, if any, is shifted to the right to remove the data previously at the cursor position. If there is no data located to the left of the cursor position, the data at the cursor position becomes null.

- Cursor right key

The cursor right key moves the cursor to the right if there is previously entered data to the right. If there is no previously entered data to the right, or if the end of the field is reached, wraparound to the first position of the field occurs.

If the field is displayed in reverse order of entry, wraparound occurs to the position preceding the leftmost character previously entered. You can enter data regardless of the cursor position. If the cursor has been moved from the rightmost position, data is inserted at the cursor position. Data to the right of that cursor position is not shifted. Any data at or to the left of that cursor position is shifted left.

- Cursor left key

The cursor left key moves the cursor to the left if it is not at the beginning of the field. If it reaches the beginning of the field, wraparound occurs to the rightmost position in the field that contains previously entered data. If there is no previously entered data, the cursor remains at the first position of the field.

If the field is displayed in reverse order of entry, wraparound occurs from the leftmost position containing previously entered data to the rightmost position of the field.

- Insert key

The Insert key is a toggle. Data keys following the Insert key (until it is pressed again) are inserted into the field at the cursor position, the cursor moves to the right, and data to the right of the cursor's previous position is moved to the right.

If the field is displayed in reverse order of entry, the Insert key has no effect. Data to the left of the cursor is moved to the left as data is inserted at the cursor position.



- Delete key

The Delete key causes the data at the cursor position to be deleted. All data to the right of the cursor is moved to the left to occupy the vacated position. The cursor remains in the same position.

If the field appears in reverse order of entry, data to the left of the cursor is moved to the right to occupy the vacated position.

**Note:** The cursor up, cursor down, PgUp, PgDn, Home, and End keys are processed as any other configurable function keys.

## Customization of Message Display (Enhanced Full-Screen mode only)

The I/O processor writes data to the display in three general categories: state prompt messages, error messages, and display-as-keyed data. If the enhanced full-screen support is not selected, this data is always written to the top left 2x20 area of the video display. If the enhanced full-screen support is selected and the state definition specifies that the display fields are defined in the function code definitions, the data written by the I/O processor may be customized in the following ways:

- Location

You specify the row and column of the upper left character of the message area. In the case of a bordered message, this would be the location of the top left corner character.

- Width of display area

You may specify the width of the display area. If a border was specified (see below), this width includes the 2 positions occupied by the left and right border characters. The specified display width may be narrower or wider than the actual message. If it is narrower, then the message text is clipped to the display width (if bordered, it is clipped to within the borders). If the display width is specified as being wider than the message, then the unwritten area of the display width is written with blanks.

- Border

You may specify that a border surrounds the message. You specify the border characters as eight hex values for the eight parts of the border: top left corner, top, top right corner, left, right, bottom left corner, bottom, bottom right corner.

- Attributes (colors, blink, etc.)

You may specify a hex attribute byte to be used for the message area. This byte must be defined in the Feature A attribute format of the video driver. See the video section in the *4680 BASIC: Language Reference* for a description of the Feature A attribute format.

- Single or Double Line Formatting

You may specify whether the message text is to be displayed in a 2x20 format or a single line format. Although messages are no longer necessarily displayed in a 2x20 format, the method of defining this data remains unchanged. That is, the IST Build Utility presents you with a 2x20 box in which to specify the message text.

If the single line format is selected, the I/O processor concatenates the line-1 text and the line-2 text such that there is exactly one blank between the last non-blank character preceding column 20 and the first non-blank character at or after column 20. This is so that messages formatted for 2x20 display appear correctly without requiring redefinition.

- Centering

You may specify that the message text is to be centered within the defined display area. The centering option is ignored for display-as-keyed messages and 2x20 formatted messages.

**Note:** Customization information is specified in the Common Information section of the State Table.

There is one specification for prompt messages, one specification for error messages, and one specification for display-as-keyed data associated with non-full-screen states ( for full-screen states, there are separate display-as-keyed specifications for each field in each full-screen state). The prompt and error message specifications are in effect for all states. The display-as-keyed specification is in effect for all non-full-screen states.

The I/O processor does not save and restore the areas of the screen to which it writes data. It is the application's responsibility to clear any undesirable residual data from the screen. The I/O processor writes to the display according to information in the State Table. The State Table part of the application and the executable part of the application should be developed to work in harmony with each other.

If the display-as-keyed area is a 2x20 box at the top left of the display with default attributes and with no border, no change to this area occurs until the operator starts keying in a non-full-screen state (no change from the behavior when enhanced full-screen support is not used). If the display-as-keyed area is anything other than the default 2x20 box, it is initialized when an unlock to a non-full-screen state occurs (the same behavior as fields of a full-screen state). This implementation allows you to add full-screen function to an application that was previously non-full-screen and preserve the behavior of parts of the application that were not changed, but improve the behavior of parts that are changed.

Of concern to the application programmer is the fact that all unwritten positions of the defined display area are written with blanks in order for their attributes to be visible. If the display-as-keyed area and the prompt area are both configured to occupy the same 2x20 area on the screen, the display-as-keyed area will overlay the prompt.

As an example, consider implementing a 2x30 area to be used for both prompts and display-as-keyed data. You could configure the prompt messages as single-line, 30 characters wide, originating at row 1 column 1, and the display-as-keyed messages as single line, 30 characters wide, originating at row 2 column 1. The display-as-keyed area no longer interferes with the prompt area. Note that for this case, the asterisks defining the location of keyed data must not be placed beyond column 10 of the 2nd line, as this is offset 30 into the defined display area. Any data that spills out of the defined display area is clipped (not displayed).

**An Example: Implementing a Shared Message Line:** Assume that you are using a 16x60 display, and that you want to use the bottom line of the video for both application and I/O processor messages. Using the IST Build Utility, define the error messages location and attributes as follows:

- Display at row 16, column 1.
- Format for single line display.
- The display area is 60 characters wide.
- The text is to be centered within the display area.

When the I/O processor detects an error, it displays the message text on the defined message line. There is a problem here, however. The error message is not cleared from the message line after the error condition has been corrected. The solution is to define a blank message to be displayed when a function code is received without error. Use the IST Build utility to define this message for each function code definition (it is necessary to type spaces over the underscores to create a blank message). The function code for an input field is received when a tab or motor key is pressed. This is a reasonable time to clear the message.

The application is free to write to the same message line. To prevent the I/O processor from writing to the message line at the same time as the application, the application should ensure that the I/O processor is locked. The application does a PUTLONG to the video display to set the color, then a LOCATE to row 16 column 1, then a WRITE to the video. Since the operator does not need to be aware of the source of the message, the application and the I/O processor form a more integrated solution than that which is possible without using the enhanced full-screen support.

## Large Input Sequences (Enhanced Full-Screen mode only)

When the enhanced full-screen support is not selected, the maximum number of input fields per state is limited to 10 (not including the status field). The maximum has been raised to 127 when enhanced full-screen support is selected. Relative positions 1 through 127 may be defined. The actual number of relative positions that are received by an application on a read of the I/O processor depends on the highest relative position defined for any state in the input sequence table. If the highest number defined is 10 or less, then there will be 10 relative positions in the input sequence (not including the status field). Otherwise, the number of relative positions in the input sequence will be equal to the highest relative



position defined for any state. This is the number of relative positions that an application will receive for EACH read of the I/O processor, regardless of which state is current. Specifying too few or too many variables on a READ FORM statement is an error. For this reason, READ LINE is now the recommended method of reading from the I/O processor. The 3-character ASCII representation of the number of relative positions contained in the input sequence is available in the status field. See *Status Field Extensions* below for details. The buffer size specified in the BASIC OPEN statement previously limited the input sequence buffer size to 200 bytes. This limit has been removed. There is no artificial limit imposed on the buffer size.

**EXAMPLE 1:** An input sequence table defines 3 states. State 1 defines function codes with relative positions 1,2,3,4, and 5. State 2 defines function codes with relative positions 1,6, and 9. State 3 defines function codes with relative positions 3 and 4.

On each read of the I/O processor, regardless of which state is current, the I/O processor will provide an input sequence containing 10 relative positions (not including the status field).

**EXAMPLE 2:** An input sequence table defines 3 states. State 1 defines function codes with relative positions 1,2,3,4, and 5. State 2 defines function codes with relative positions 1,6, and 9. State 3 defines function codes with relative positions 9,10,11,12 and 13.

On each read of the I/O processor, regardless of which state is current, the I/O processor will provide an input sequence containing 13 relative positions (not including the status field).

## PDF417 Labels

PDF417 labels allow the system to handle driver's license data from each state in the USA.

- All driver's licenses that have a scanner label on the back have that label encoded in PDF417.
- There is no standard for the content and format of the driver's license data. Each of the 50 states may be different.
- The amount of data encoded in the code may be as much as 2710 bytes of data. However, some scanners may support less than 2710 bytes.
- Some or all of the data encoded in the label may be encrypted. There is no standard encryption method. Each of the 50 states may be different.
- Each state may change the format and content of their own driver's licenses at any time.
- There may be more than one driver's license PDF417 format and content in circulation in any single state at any one time.

Because of the amount of data associated with the PDF417 label and the varying nature of the content and format, the 4690 OS will not parse any of the PDF417 data. The 4690 OS will pass the data to the application in its raw form.

**Overview of I/O Processor and Java I/O Processor Changes:** When a PDF417 label is scanned, the 4690 OS I/O Processor or Java I/O Processor generates an input sequence buffer with a single data field containing the length of the PDF417 label data, along with the specified function code. Because the 4690 OS is passing the raw PDF417 label data to the application, the application does not get the label data from the input sequence buffer. The 4690 OS writes the PDF417 label data to a new application created pipe, the PDF417 label data pipe.

**Overview of JavaPOS Changes:** Applications that use 4690 JavaPOS scanner controls for scanner input must be updated to recognize the PDF417 label type, SCAN\_SDT\_PDF417, and decrypt (if necessary) and parse the PDF417 label data.

**PDF417 Scanner Configuration:** Scanners must be USB attached, support PDF417 labels, and conform to the *Universal Serial Bus OEM Pointof-Sale Device Interface Specification*. This document can be requested through the Techline found on the Toshiba support site.

1. Configure the scanner to be used as a 4690 OS USB scanner (manufacturer dependent procedure).

2. Configure the scanner to enable PDF417 labels (manufacturer dependent procedure).

The HHP 4600 Scanner, vendor id = 0536, product id = 04c3, must be at ec level = 003f for the 4690 OS to provide PDF417 label support.

## **How to use PDF417 Labels with applications that use the I/O Processor or Java I/O Processor**

**Update the Label Format Table:** Using 4690 OS configuration, the POS application's Label Format Table must be updated to add a new PDF417 label format definition. This definition allows only 1 data field of length 4 and the associated function code. Specify the function code that the POS application will process as a PDF417 label.

**Update the Input State Table:** Using 4690 OS configuration, the POS application's Input State Table must be updated to include the function code defined as a PDF417 label in the application's Label Format Table.

### **Update the POS Application: Process the PDF417 Input Sequence Buffer**

The POS application's processing of scanner input sequence buffers must be updated to recognize the function code defined in the Label Format Table as a PDF417 label and process the unique format of the PDF417 label input sequence buffer data. The format of the PDF417 label input sequence buffer is a single data field containing the length of the PDF417 label data as 4 ASCII bytes. For example, if the length of the PDF417 label data is 27 bytes, the input sequence buffer will contain 30303237.

### **Process the PDF417 Label Data**

The POS application must be updated to create the PDF417 label data pipe. The PDF417 label data pipe is a pipe named "pi:ioplxxx" (where 'xxx' is the terminal number). The size of the pipe should be at least as large as the largest expected PDF417 label. The maximum PDF417 label data size supported by the 4690 OS is 2710 bytes.

The POS application must be updated to read the PDF417 label data from the PDF417 label data pipe.

The 4690 OS write to pipe operation will block until there is enough room in the pipe for the data to be deposited. This means that the POS application must react in a timely fashion to read the data from the pipe, so that the pipe is prepared to receive the next PDF417 label.

It is up to the POS application to decrypt (if necessary) and parse the PDF417 label data.

If the application has not created the PDF417 label data pipe, or the PDF417 label type has not been defined in the Label Format Table, when a PDF417 label is scanned, the 4690 OS will sound an error tone to indicate that the label was not successfully processed.

Refer to the *4690 OS Version 5: Programming Guide* for information on how to create and read from 4690 pipes.

---

## **Magnetic Stripe Reader Driver (MSR)**

s

This section describes the single-, dual-, three-, and JIS-II magnetic stripe reader drivers and provides guidelines for using them.

## Characteristics of the Single-Track Magnetic Stripe Reader

The single-track MSR has the following characteristics:

- The single-track MSR attaches to the top of the keyboard and connects to the keyboard by a cable.
- The MSR can read data encoded on a card (credit card or badge) as the card is passed through a slot in the reader. The single-track MSR reads track 2 data as specified in the following standards:
  - “The American National Standards Institute (ANSI) Specifications for Magnetic-Stripe Encoding for Credit Cards”, ANSI X4.16-1983
  - “The American National Standards Institute (ANSI) Specifications for Credit Cards”, ANSI X4.13-1983.
- The data available to your application consists of the account number field, a separator character, and a discretionary data field. The account number can be a maximum of 19 characters. The discretionary data field can be up to 36 characters. The total number of characters occupied by the account number field, the separator character, and the discretionary data field cannot exceed 37.

## Characteristics of the Dual-Track Magnetic Stripe Reader

The dual-track MSR has the following characteristics:

- The dual-track MSR attaches to the top of the keyboard or connects directly to slot 5B on the back of the base unit.
- There are two models available: one reads tracks 1 and 2 data and the other reads tracks 2 and 3 data.
- The returned data consists of a buffer of 144 bytes. The first 37 bytes of data from track 2 are in the same format as the single-track MSR (the 37 bytes are padded with X'00' characters if needed). The data from track 2 is followed by a one-byte field containing the length of the actual data in the first 37 bytes (or X'FF' to indicate an error on track 2). Following that field is a one-byte field with the length of the data from track 1 or 3 (or X'FF' for error) followed by 105 bytes of data from track 1 or 3 (the 105 bytes are padded with X'00' if needed).
- When receiving data, the dual-track MSR formats track 1 data as alphanumeric using only the lower 6 bits of each byte. The maximum number of data characters on track 1 is 80 characters. Track 3 data is numeric only and has the same format as track 2 binary coded decimal (BCD). The maximum number of data characters on track 3 is 105 characters.

X'FF' in either length field is the same as return code 80A5000B for the track with the error. The 80A5000B code is returned if both tracks contain read errors. If an error is encountered on track 2, and track 1 or 3 is valid, the track 2 data starts with 1 byte of X'00' followed by the separator X'0D' then padded with X'00' to 37 bytes.

## Characteristics of the Three-Track Magnetic Stripe Reader

The three-track MSR can read all or any combination of the three tracks.

- Track 1 data is alphanumeric. The driver returns a maximum of 98 data characters from track 1.
- Track 2 data is numeric only. The driver returns a maximum of 46 data characters from track 2. However, if operating in single- or dual-track mode, the maximum number of data characters returned is 37.
- Track 3 data is numeric only. The driver returns a maximum of 139 data characters from track 3.

## Characteristics of the JIS-II Magnetic Stripe Reader

The JUCC MSR has JIS-II and JIS-I track-2. The JUCC MSR can read all or any combination of the two tracks.

- JIS-II track data is alphanumeric. The driver returns a maximum of 69 data characters from the JIS-II track.
- Track 2 data is numeric only. The driver returns a maximum of 37 data characters from track 2. However, if operating in single- or dual-track mode, the maximum number of data characters returned is 37.

## Data Formats and Error Reporting

The data formats for the various MSR devices supported by the JUCC MSR are:

- Single-track MSR

Single-track MSRs support reading data from track 2 only. The return code from the READ statement provides the actual number of bytes read. See Format 01 in Figure 5 on page 87.

If an error occurs, the first byte of the returned buffer contains a X'00', followed by the separator character X'0D'. The remainder of the buffer is padded with X'00'

- Dual-track MSR

- Dual-track MSRs read data from track 2 and either track 1 or track 3. The data is returned in a single buffer. The return code of the READ statement provides the size of the buffer. See Format 02 in Figure 5 on page 87.

- The dual-track MSR driver supports single-track emulation. The single-track MSR description applies when operating in this mode.

- The length field of each track contains the actual number of characters read from that track. If a read error occurs on either track, but not both, the length field for the track on which the error occurred has a value of X'FF', and the buffer is padded with X'00'. If a read error occurs on both tracks, the return code from the READ statement is 80A5000B, and the entire buffer is padded with X'00'. See the *4690 OS: Messages Guide* for a description of the 80A5000B return code.

- Three-track MSR

Three-track MSRs read data from all tracks.

If the device is configured for single-track emulation (track 2 only), or as a dual-track device (tracks 1 and 2 or tracks 2 and 3), the amount and format of the data is identical to that previously described.

The three-track device provides additional individual track capacities and capabilities. The various formats are described in Figure 5 on page 87.

Error reporting is done the same as with the dual-track MSR. If one or more, but not all, configured tracks report a read error, the length field for each track in error has a value of X'FF', and the buffer for each track is padded with X'00'. If a read error occurs on all configured tracks, the return code from the READ statement is 80A5000B, and the entire buffer is padded with X'00'. See the *4690 OS: Messages Guide* for a description of the 80A5000B return code.

- JUCC MSR

JUCC MSRs read data from track 2 and the JIS-II tracks.

Error reporting is performed the same as with the dual-track MSR. If one or more, but not all, configured tracks report a read error, the length field for each track in error has a value of X'FF', and the buffer for each track is padded with X'00'. If a read error occurs on all configured tracks, the return code from the READ statement is 80A5000B, and the entire buffer is padded with X'00'. See the *4690 OS: Messages Guide* for a description of the 80A5000B return code.

## Format of the Data in the Applications Buffer

Format 01 - Single-track - Track 2 data returned

T2 Data
---------

37 bytes maximum

Format 02 - Multi-track - Track 2 data and Track 1 or 3 data returned

T2 Data	T2 Len	Tn Len	Tn Data
---------	--------	--------	---------

37 bytes      1 byte      1 byte      n bytes  
n = Track 1 or 3 (98 or 139 respectively)

Format 04 - Single-track - Track 1 data returned

T1 Len	T1 Data
--------	---------

1 byte      98 bytes

Format 08 - Single-track - Track 3 data returned

T3 Len	T3 Data
--------	---------

1 byte      139 bytes

Format 10 - Multi-track - Tracks 1 and 3 data returned

T1 Len	T1 Data	T3 Len	T3 Data
--------	---------	--------	---------

1 byte      98 bytes      1 byte      139 bytes

Format 20 - Multi-track - All tracks data returned

T1 Len	T1 Data	T2 Len	T2 Data	T3 Len	T3 Data
--------	---------	--------	---------	--------	---------

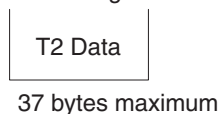
1 byte      98 bytes      1 byte      46 bytes      1 byte      139 bytes

Tn = track number

Figure 5. Multi-Track Reader Data Formats (except JUCC MSR)

## Format of the Data in the Applications Buffer

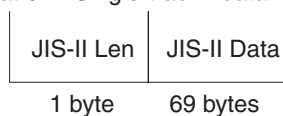
Format 01 - Single-track - Track 2 data returned



Format 02 - Multi-track - Track 2 data returned



Format 04 - Single-track - data returned



Tn = track number

Figure 6. Multi-Track Reader Data Formats (including JUCC MSR)

Table 10 on page 88 shows the applicable formats of returned data as a function of MSR configuration.

Table 10. Applicable Formats of Returned Data

Device Type	Configuration		Format
	Number of Tracks	Which Tracks	
Dual-Track	1	2	01
Dual-Track	2	1 and 2	02
Dual-Track	2	2 and 3	02
Three-Track	1	1	04
Three-Track	1	2	01
Three-Track	1	3	08
Three-Track	2	1 and 2	02
Three-Track	2	2 and 3	02
Three-Track	2	1 and 3	10
Three-Track	3	1, 2, and 3	20
JUCC MSR	1	1	04
JUCC MSR	1	2	01
JUCC MSR	2	1 and 2	02

## Restrictions of Single- and Multi-Track MSRs

Only one is allowed per terminal. Mod1 and Mod2 terminal pairs cannot have different types of MSRs attached.

## Functions Your Application Performs

Your application program can perform the following functions with the MSR:

- Allow or disallow data to be read from a card
- Wait until data is available
- Read card data

- Determine data-allowed or data-disallowed status
- Determine format of the data returned by the MSR

## Accessing the MSR

Use the OPEN statement to gain access to the MSR driver.

The CLOSE statement ends communication with the MSR. CLOSE locks the MSR and discards any data available.

## Preparing to Receive Data from the MSR

The MSR can be in two states, *locked* and *unlocked*. In the locked state, data cannot be read. In the unlocked state, data can be read. Initially the MSR is in the locked state. You must issue an UNLOCKDEV statement to put the MSR in the unlocked state. After being read by the MSR, data is available but not passed to the application until a READ LINE statement is run. If your application issues an UNLOCKDEV statement when the MSR has data available, the data is discarded.

The MSR changes from unlocked to locked when one of the following occurs:

- Valid data is read from a card.
- Your application issues a LOCKDEV statement.
- Your application ends communication with the MSR by a CLOSE statement.

Issue the UNLOCKDEV statement each time you prepare to read data from the MSR.

## Waiting for Received Data

Your application should issue a WAIT statement to wait for data available from the MSR. The WAIT statement allows the application to wait for input from multiple devices. These devices include a timer to indicate that no input was received during a specific time. When valid data is available from the MSR, the statement following the WAIT is performed. Following the WAIT, use the EVENT% function to determine if data is available to be read from the MSR. If an error occurs during a WAIT, control is given to your ON ERROR routine.

## Receiving Data

Issue a READ LINE statement to receive data from the MSR. READ LINE is a synchronous operation. If valid data is available from the MSR, the READ LINE is completed, and the variable specified on the READ LINE statement contains the data. If an error occurs, control is given to the ON ERROR routine.

The driver validates the buffer size passed to it. If the buffer size is insufficient to handle all potential data for the active configuration, the driver returns 80A50004. See the *4690 OS: Messages Guide* for a detailed description of the 80A50004 return code.

After good data is received, the MSR is locked.

## Data Characteristics Common to all MSRs

The data characters in the application's buffer are in BCD format. Only the characters 0 through 9 and a separator character can appear in the buffer. Zero is 00H, nine is 09H, and the separator is 0DH.

Parity checking and other data used by the hardware are not passed to the application. The return code from the READ LINE statement contains the amount of data returned.

The characteristics specific to single-track or multi-track MSRs are:

- Single-track MSRs
  - The total number of bytes returned to the application cannot exceed 37.
- Multi-track MSRs

- The maximum number of data characters on track 1 is 102.
- The maximum number of data characters available to the application from track 2 is increased to 46 when operating a three-track device in specific configurations (see Figure 5 on page 87).
- The maximum number of data characters on track 3 is 103 for a dual-track device and 139 for a three-track device.

## Disallowing MSR Data

Issue the LOCKDEV statement to prevent the MSR from receiving card data. If data is already available when you issue the LOCKDEV statement, the data is discarded.

## Determining Status of the MSR

Use the GETLONG statement to make the following determinations:

- The state of the MSR: Locked or Unlocked
- Which of tracks 1 or 3 is connected for a dual-track MSR
- The format of the returned data for a three-track MSR

### Integer Format for Single-Track MSR Driver

The integer represents information in the form *RRRRRLL*. *RR*, *RR*, *RR*, and *LL* represent one of the four bytes. The *RRRRR* bytes are reserved for system use.

The *LL* bytes represent the following state information:

*LL* = 0 if the MSR is unlocked.

*LL* = 1 if the MSR is locked.

### Integer Format for Dual-Track MSR Driver

The integer represents information in the form of *RRSSRLL*, where *RR*, *SS*, *RR*, and *LL* each represent one of the four bytes. The only difference between the single and dual-track data is *SS*, which represents the status information. The *RR* bytes are reserved for system use.

The *LL* byte represents the following state information:

*LL* = 0 if the MSR is unlocked.

*LL* = 1 if the MSR is locked.

### Integer Format for Three-Track MSR Driver

The integer represents information in the form *RRSSFFLL*, where *RR*, *SS*, *FF*, and *LL* each represent one of the four bytes. The *RR* bytes are reserved for system use.

The *LL* byte represents the following state information:

*LL* = 0 if the MSR is unlocked.

*LL* = 1 if the MSR is locked.

The *SS* byte represents the following status information:

**Bit 0** = 0 RESERVED  
**Bit 1** = 0 RESERVED  
**Bit 2** = 1 Track 1 Enabled  
**Bit 3** = 1 Track 2 Enabled  
**Bit 4** = 1 Track 3 Enabled  
**Bit 5** = 1 EC Level Response  
**Bit 6** = 0 RESERVED  
**Bit 7** = 0 RESERVED

The *FF* byte represents the following formatting information:

**Bit 0** = 1 Format 01  
**Bit 1** = 1 Format 03  
**Bit 2** = 1 Format 04



**Bit 3 =** 1 Format 08  
**Bit 4 =** 1 Format 10  
**Bit 5 =** 1 Format 20  
**Bit 6 =** 0 RESERVED  
**Bit 7 =** 0 Dual-Track Device  
**Bit 7 =** 1 Three-Track Device

## Integer Format for JUCC MSR Driver

The integer represents information in the form *RRSSFFLL*, where *RR*, *SS*, *FF*, and *LL* each represent one of the four bytes. The *RR* bytes are reserved for system use.

The *LL* byte represents the following state information:

*LL* = 0 if the MSR is unlocked.

*LL* = 1 if the MSR is locked.

The *SS* byte represents the following status information:

**Bit 0 =** 0 RESERVED  
**Bit 1 =** 0 RESERVED  
**Bit 2 =** 0 RESERVED  
**Bit 3 =** 1 Track 2 Enabled  
**Bit 4 =** 0 RESERVED  
**Bit 5 =** 1 JIS-II Track 2 Enabled  
**Bit 6 =** 1 EC Level Response  
**Bit 7 =** 0 RESERVED

The *FF* byte represents the following formatting information:

**Bit 0 =** 1 Format 01  
**Bit 1 =** 1 Format 02  
**Bit 2 =** 1 Format 04  
**Bit 3 =** 0 RESERVED  
**Bit 4 =** 0 RESERVED  
**Bit 5 =** 1 RESERVED  
**Bits 6 and 7 =** 1 JUCC MSR

## Single-Track MSR Example

This example reads data from a card passed through the MSR. The data is converted and displayed. The example processes three card reads before ending.

```

%ENVIRON T
! Declare integers.
INTEGER*2 DISPLAY
INTEGER*2 MSREADER
! Convert data to display characters.
FUNCTION CONVERT.TO.HEX$(THE.SUM%)
  IF THE.SUM% > 9 THEN \
    THE.SUM%=THE.SUM%+55\
  ELSE \
    THE.SUM%=THE.SUM%+48
  CONVERT.TO.HEX$=CHR$(THE.SUM%)
  EXIT FUNCTION
END FUNCTION

DISPLAY = 1
MSREADER = 2
ON ERROR GOTO ERR.HNDLR
! Open the display.
OPEN "ANDISPLAY:" AS DISPLAY
! Write a greeting and wait 2 seconds.
CLEARS DISPLAY
  
```

```

WRITE #DISPLAY; "MS READER SAMPLE"
WAIT;2000
! Open the magnetic stripe reader driver.
OPEN "MSR:" AS MSREADER
! Loop and read three cards.
FOR I% = 1 to 3
  UNLOCKDEV MSREADER
  ! Display instructions.

  CLEARS DISPLAY
  WRITE #DISPLAY;"PASS A CARD THROUGH"
  MSRWAIT:
  ! Wait for input 5 seconds.
  WAIT MSREADER;5000
  ! You would normally test for
  ! a timeout condition.
  ! Read the available input data.
  READ #MSREADER; LINE MSRDATA$
  ! Convert the input to characters that can be displayed
  DISPL.DATA$ = " "
  ! Loop through the input data and convert
  ! each character.
  FOR J% = 1 to LEN(MSRDATA$)
    A$=MID$(MSRDATA$,J%,1)
    THE.SUM% = ASC(A$)
    A$=CONVERT.TO.HEX$(THE.SUM%)
    DISPL.DATA$ = DISPL.DATA$ + A$
  NEXT J%
  ! Display input data and wait 2 seconds.
  CLEARS DISPLAY
  WRITE #DISPLAY;DISPL.DATA$
  WAIT;2000
NEXT I%
CLEARS DISPLAY
WRITE #DISPLAY; "END OF SAMPLE"
CLOSE DISPLAY,MSREADER
STOP

ERR.HNDLR:
! Prevent recursion.
ON ERROR GOTO END.PROG
! Determine error type and perform appropriate
! recovery and resume steps.
END.PROG:
STOP
END

```

## Dual-Track MSR Example

This example reads data from a card passed through the MSR. The data is converted and displayed. The example processes three card reads before ending.

```

%ENVIRON T
! Declare variables.
INTEGER*2 DISPLAY
INTEGER*2 MSR2.READER
STRING MSR2.DATA
STRING MSR2.DATA.TRACK2
STRING MSR2.DATA.TRACK13
INTEGER*2 L1,L2
DISPLAY=1
MSR2.READER=2
ON ERROR GOTO ERR.HNDLR
! Open the display.
OPEN "ANDISPLAY:" AS DISPLAY
! Write greeting and wait 2 seconds.
CLEARS DISPLAY
WRITE #DISPLAY; "MSR2 SAMPLE"
WAIT;2000

```

```

! Open the MSR driver.
OPEN "MSR:" AS MSR2.READER
! Loop and read 3 cards.
FOR I%=1 TO 3
  UNLOCKDEV MSR2.READER
  ! Display instructions.
  CLEARS DISPLAY
  WRITE #DISPLAY; "PASS A CARD THROUGH ",I%
  MSRWAIT:
  ! Wait for 5 seconds.
  WAIT MSR2.READER;5000
  ! You would normally test for a timeout condition.
  ! Read the available input data.
  READ #MSR2.READER; LINE MSRDATA$
  ! Get the length of the track 2 data (255 indicates error).
  L1=ASC(MID$(MSRDATA$,38,1))
  ! Get the length of the track 1 or 3 data (255 indicates error).
  L2=ASC(MID$(MSRDATA$,39,1))

  ! Process track 2 data.
  IF L1 = 255 THEN \
    BEGIN
    ! Display error message.
    CLEARS DISPLAY
    WRITE #DISPLAY; "TRACK 2 ERROR"
    WAIT;3000
    ENDIF \
  ELSE \
    BEGIN \
      MSR2.DATA.TRACK2 = MID$(MSRDATA$,1,L1)
      ! Here you would convert the input to
      ! characters and display it.
      ENDIF
    ! Process track 1/3 data.
    IF L2 = 255 THEN \
      BEGIN
      ! Display error message.
      CLEARS DISPLAY
      WRITE #DISPLAY; "TRACK 1/3 ERROR"
      WAIT;3000
      ENDIF \
    ELSE \
      BEGIN \
        MSR2.DATA.TRACK13 = MID$(MSRDATA$,40,L2)
        ! Here you would convert the input to
        ! characters and display it.
        ENDIF
      NEXT I%
      CLEARS DISPLAY
      WRITE #DISPLAY; "END OF SAMPLE"
      CLOSE MSR2.READER,DISPLAY
      STOP
    ERR.HNDLR:
    ! Prevent recursion.
    ON ERROR GOTO ERR.EXIT
    ! Determine error type and
    ! perform appropriate recovery
    ! and resume steps.
    ERR.EXIT:

  STOP
END

```

## Three-Track MSR Example

The following example reads data from a card passed through the MSR. The data is converted and displayed. The example processes three cards before ending.

```

%ENVIRON T
! Declare variables.
INTEGER*2 DISPLAY
INTEGER*2 MSR2.READER
INTEGER*2 L1, L2, L3
INTEGER*4 CONFIG,MSRTYPE,MSRFMT,MSRSTA
STRING  MSRDATA$,MSRDATA1$,MSRDATA2$,MSRDATA3$,DISPLINE$
STRING  MSRSTA$
DISPLAY=1
MSR2.READER=2
ON ERROR GOTO PROGERR
! Open the display.
OPEN "ANDISPLAY:" AS DISPLAY
! Write greeting and wait 2 seconds.
CLEARS DISPLAY
DISPLINE$="MSR3 SAMPLE"
WRITE #DISPLAY; DISPLINE$
WAIT;2000
! Open the MSR driver.
OPEN "MSR:" AS MSR
! Get device information.
CONFIG=GETLONG(MSR)
! Isolate device type.
MSRTYPE=CONFIG AND 00008000H
MSRTYPE=SHIFT(MSRTYPE,8)
! Here you would determine if operating with a dual-track
! or 3-track device.
! Isolate data format.
MSRFMT=CONFIG AND 00007F00h
MSRFMT=SHIFT(MSRFMT,8)
! Here you would determine the format of the data
! to be received.
! Isolate status.
MSRSTA=CONFIG AND 00FF0000H
MSRSTA=SHIFT(MSRSTA,16)
! Here you would determine the status of the device.
! Loop and read three cards.
FOR I%=1 TO 3
  UNLOCKDEV MSR
  CLEARS DISPLAY
  DISPLINE$= "PASS A CARD THROUGH",I%
  WRITE #DISPLAY; DISPLINE$
  MSRWAIT:
  ! Wait for 5 seconds.
  WAIT MSR;5000
  ! You would normally test for a timeout condition.
  ! Read the available input data.
  READ #MSR; LINE MSRDATA$
  ! Get the length of the data on each track - this example
  ! assumes all three tracks are configured
  L1=ASC(MID$(MSRDATA$,1,1))
  L2=ASC(MID$(MSRDATA$,100,1))
  L3=ASC(MID$(MSRDATA$,147,1))
  ! Process track 1 data.
  IF L1 = 255 THEN \
    BEGIN
    ! Display track error message.
    CLEARS DISPLAY
    DISPLINE$= "TRACK 1 ERROR"
    WRITE #DISPLAY; DISPLINE$
    ENDIF \
  ELSE \
    BEGIN \
      MSRDATA1$ = MID$(MSRDATA$,2,L1)
      ! Here you would convert the input to
      ! displayable characters and display it.
      ENDIF

```

```

! Process track 2 data.
IF L2 = 255 THEN \
    BEGIN
! Display track error message.
    CLEARS DISPLAY
    DISPLINE$= "TRACK 2 ERROR"
    WRITE #DISPLAY; DISPLINE$
    ENDIF \
ELSE \
    BEGIN \
        MSRDATA1$ = MID$(MSRDATA$,101,L2)
! Here you would convert the input to
! displayable characters and display it.
    ENDIF
! Process track 3 data.
IF L3 = 255 THEN -
    BEGIN
! Display track error message.
    CLEARS DISPLAY
    DISPLINE$= "TRACK 3 ERROR"
    WRITE #DISPLAY;DISPLINE$
    ENDIF \
ELSE \
    BEGIN \
        MSRDATA1$ = MID$(MSRDATA$,148,L3)
! Here you would convert the input to
! displayable characters and display it.
    ENDIF
NEXT I%
CLEARS DISPLAY
DISPLINE$= "END OF SAMPLE"
WRITE #DISPLAY; DISPLINE$
CLOSE MSR
STOP
PROGERR:
! Prevent recursion.
ON ERROR GOTO ERR.EXIT
! Determine error type and
! perform appropriate recovery
! and resume steps.
ERR.EXIT:

STOP
END

```

## JUCC MSR Example

The following example reads data from a card passed through the MSR. The data is converted and displayed. The example processes three cards before ending.

```

%ENVIRON T

    STRING ErrFlag$
    INTEGER*2 MSR
    INTEGER*2 A0%, A1%, A2%, A3%, A4%, A5%, A6%, A7%
    INTEGER*4 CONFIG%, MSRTYPE%, HX%

! Convert data to display characters.
FUNCTION CONVERT.TO.HEX$(THE.SUM%)
    IF THE.SUM% > 9 THEN \
        THE.SUM%=THE.SUM%+55 \
    ELSE \
        THE.SUM%=THE.SUM%+48
    CONVERT.TO.HEX$=CHR$(THE.SUM%)
    EXIT FUNCTION
END FUNCTION

```

```

FUNCTION PRINTER
  OPEN STATIONS$ AS NUMA
  CLEARS DISPLAYN
  EXIT FUNCTION
END FUNCTION

FUNCTION CLOSEA
  TCLOSE NUMA
  CLEARS DISPLAYN
  CLOSE NUMA
  EXIT FUNCTION
END FUNCTION

!-----
!
! SUB-PROGRAM
!
!-----

SUB OPMSG (MSG$)
  STRING MSG$
  CLEARS DISPLAYN
  LOCATE #DISPLAYN; 1, 1           ! POSITION CURSOR
  WRITE #DISPLAYN; MSG$           ! WRITE MESSAGE
  WAIT; 3000                       ! WAIT ALMOST FOREVER
END SUB

SUB ASYNCSUB(RFLAG,OVER)
  INTEGER*2 RFLAG
  STRING OVER

  ! Set up synchronous error handler for async subprogram.
  ! ON ERROR GOTO END.PROG
  ErrFlag$="TRUE"
  ERR$ = ERR
  CLEARS DISPLAYN

  ! See the error recovery example for the ON ASYNC ERROR CALL statement.

  HX% = ERRN
  ERRFX$ = ""
  FOR S% = 28 TO 0 STEP -4
    SX% = SHIFT(HX%,S%)
    THE.SUM% = SX% AND 000FH
    IF THE.SUM% > 9 THEN BEGIN
      THE.SUM%=THE.SUM%+55
    ENDIF ELSE BEGIN
      THE.SUM%=THE.SUM%+48
    ENDIF
    Z$=CHR$(THE.SUM%)
    ERRFX$ = ERRFX$ + Z$
  NEXT S%

  CLEARS DISPLAYN
  WRITE #DISPLAYN ;"Asyn ERR = " + ERR
  LOCATE #DISPLAYN ; 2, 1
  WRITE #DISPLAYN ;"ERRN = " + ERRFX$
  WAIT;3000
  CALL OPMSG("STATION = " + STATIONS$)
  WAIT; 3000
  IF ERR$ ="RN" THEN EXIT SUB           ! UNRESUMABLE ERROR
  !RESUME

  EXIT SUB
END SUB                                     ! End Sub of ASYNCSUB

!-----

```

```

!
!   MAIN PROGRAM
!
!-----

    ON ASYNC ERROR CALL ASYNCSUB
    ON ERROR GOTO END.PROG

    DISPLAYN = 1
    MSR = 9

    OPEN "VDISPLAY:" AS DISPLAYN

!-----
! DUAL-TRACK: RR SS FF LL    each represent one of the four bytes
!-----
    CALL OPMSG ("START GETLONG DUAL-TRACK MSR TEST")
    ErrFlag$="FALSE"
    CALL OPMSG ("OPEN THE MSR")
    OPEN "MSR:" AS MSR
    CALL OPMSG ("UNCLOCKDEV THE MSR")
    UNLOCKDEV MSR

    CONFIG%=GETLONG(MSR)
    WRITE FORM "PIC(READ CONFIG  = #####)";#DISPLAYN;CONFIG%
    WAIT;3000

    TMP1% = CONFIG% AND 000000FFH
    CLEARS DISPLAYN
    LOCATE #DISPLAYN; 1,1
    WRITE #DISPLAYN; "DUAL-TRACK: RR SS FF LL"
    LOCATE #DISPLAYN; 2,1
    WRITE FORM "PIC(READ ???X LL = #####)";#DISPLAYN;TMP1%
    WAIT;3000

    TMP2% = SHIFT((CONFIG% AND 0000FF00H), 8)
    CLEARS DISPLAYN
    WRITE FORM "PIC(READ ??X? FF = #####)";#DISPLAYN;TMP2%
    WAIT;3000

    TMP3% = SHIFT((CONFIG% AND 00FF0000H), 16)
    CLEARS DISPLAYN
    WRITE FORM "PIC(READ ?X?? SS = #####)";#DISPLAYN;TMP3%
    WAIT;3000

    TMP4% = SHIFT((CONFIG% AND FF000000H), 24)
    CLEARS DISPLAYN
    WRITE FORM "PIC(READ X??? RR = #####)";#DISPLAYN;TMP4%
    WAIT;3000

!-----
!
! Byte SS (?x??) represents status information as follows:
!
! Bit 0 = 0 RESERVED
! Bit 1 = 0 RESERVED
! Bit 2 = 0 RESERVED
! Bit 3 = 1 Track 2 Enabled
! Bit 4 = 0 RESERVED
! Bit 5 = 1 JIS-II Track 2 Enabled
! Bit 6 = 1 EC Level Response
! Bit 7 = 0 RESERVED
!
!-----

    CLEARS DISPLAYN
    MSRTYPE% = SHIFT((CONFIG% AND 00FF0000H), 16)

```

```

A0% = MSRTYPE% AND 01H
A1% = SHIFT((MSRTYPE% AND 02H), 1)
A2% = SHIFT((MSRTYPE% AND 04H), 2)
A3% = SHIFT((MSRTYPE% AND 08H), 3)
A4% = SHIFT((MSRTYPE% AND 10H), 4)
A5% = SHIFT((MSRTYPE% AND 20H), 5)
A6% = SHIFT((MSRTYPE% AND 40H), 6)
A7% = SHIFT((MSRTYPE% AND 80H), 7)

LOCATE #DISPLAYN;1,1
WRITE FORM "PIC(GETLONG STATUS INFORMATION IS = #####);#DISPLAYN;MSRTYPE%
LOCATE #DISPLAYN;2,1
WRITE #DISPLAYN;"STATES DESCRIPTION: 0:IS DIABLED 1:IS ENABLED"
LOCATE #DISPLAYN;4,1
WRITE FORM "PIC(BIT 0 = #);#DISPLAYN;A0%
LOCATE #DISPLAYN;5,1
WRITE FORM "PIC(BIT 1 = #);#DISPLAYN;A1%
LOCATE #DISPLAYN;6,1
WRITE FORM "PIC(BIT 2 = #);#DISPLAYN;A2%
LOCATE #DISPLAYN;7,1
WRITE FORM "PIC(BIT 3 = #);#DISPLAYN;A3%
LOCATE #DISPLAYN;8,1
WRITE FORM "PIC(BIT 4 = #);#DISPLAYN;A4%
LOCATE #DISPLAYN;9,1
WRITE FORM "PIC(BIT 5 = #);#DISPLAYN;A5%
LOCATE #DISPLAYN;10,1
WRITE FORM "PIC(BIT 6 = #);#DISPLAYN;A6%
LOCATE #DISPLAYN;11,1
WRITE FORM "PIC(BIT 7 = #);#DISPLAYN;A7%
WAIT;5000

```

```

CLOSE MSR

```

```

IF ErrFlag$ = "TRUE" THEN \
  BEGIN
    NUMA = 3
    STATIONS$ = "CR:"
    CALL PRINTER
    WRITE #NUMA; " TEST FAIL"
    CALL CLOSEA
    CALL OPMSG ("TEST GETLOGN DUAL-TRACK MSR FAIL")
  ENDIF \
ELSE \
  CALL OPMSG ("TEST GETLOGN DUAL-TRACK MSR PASS")

```

```

CALL OPMSG ("END OF TEST")

```

```

STOP

```

```

!-----
!  EXCEPTION HANDLER
!-----

```

```

END.PROG:
  MSG$ = ERR

```

```

ERRORTRAP:

```

```

!*: ERROR ASSEMBLY ROUTINE !*:
  ErrFlag$="TRUE"
  HX% = ERRN
  ERRFX$ = ""
  FOR S% = 28 TO 0 STEP -4
    SX% = SHIFT(HX%,S%)
    THE.SUM% = SX% AND 000FH
  
```



```

        IF THE.SUM% > 9 THEN BEGIN
            THE.SUM%=THE.SUM%+55
        ENDIF ELSE BEGIN
            THE.SUM%=THE.SUM%+48
        ENDIF

        Z$=CHR$(THE.SUM%)
        ERRFX$ = ERRFX$ + Z$

    NEXT S%

    CLEARS DISPLAYN
    WRITE #DISPLAYN ; "General ERR = " + ERR
    LOCATE #DISPLAYN ; 2, 1
    WRITE #DISPLAYN ; "ERRN = " + ERRFX$
    WAIT;3000
    IF ERR$ ="RN" THEN STOP                ! UNRESUMABLE ERROR

    RESUME

    CLOSE DISPLAYN

    STOP
END

```

---

## Printer Driver Model 2

This section describes the model 2 printer driver and provides information on accessing and using it.

### Characteristics

The printer is composed of a single bidirectional print head and two stations that provide three logical stations. The first station is called the customer receipt/document insert (CR/DI) station; the second is called the summary journal (SJ) station.

You can use the CR/DI station to print on a customer receipt or an inserted document. When a document is used, it is positioned between the customer receipt and the print head. You cannot use the CR/DI station to print on the customer receipt tape and a document at the same time.

At each station, the print head can print 38 characters on a line. It prints each character on a 7 x 8 dot matrix. Three columns of dots to the right of each character are reserved for spacing. The print line for each station measures 380 dots across.

One line feed advances the paper 11 vertical dots. A printed line becomes visible to the operator at either station after the printed line is advanced eight line feeds.

Printing one line takes approximately 0.5 seconds. One line feed takes approximately 0.075 seconds in the CR/DI station and up to 0.158 seconds in the SJ station.

The operating system supports logo printing, character printing, and check printing. This facility allows you to print any dot in the middle 300 dots of a 380-dot line at the CR/DI station. Logos cannot be printed at the SJ station. Partial line feeds are available for the CR/DI station. Each partial line feed advances the line by one dot. You can print logos taller than 8 dots by using partial line feeds to print adjoining logo patterns.

A default dot matrix is supplied for many of the available 256 code points. You can redefine the dot matrix configuration for most characters. You cannot define horizontally adjacent dots. The system prints undefined characters as a single dot. See the *4680 BASIC: Language Reference* for the definition of the default character set.

## Functions Your Application Performs

Your application program can perform the following functions with the printer:

- Write characters to any station
- Write all-points-addressable data to the CR/DI station
- Control document insertion and sensing
- Obtain status
- Wait for outstanding print lines to print

## Accessing the Printer

Use the OPEN statement to gain access to the printer device driver. The name for each print station is:

**CR:** Customer receipt station

**SJ:** Summary journal station

**DI:** Document insert station

**Note:** The colon is part of the name, and you must open each station before it will print.

The CLOSE statement ends communication with a printer station. You must issue a CLOSE statement for each print station to which your application has issued an OPEN.

## Preparing a Line To Print

You can specify the data to be printed on a line in one or more expressions. Each expression can be a string or of numeric type. The format string on the WRITE FORM statement defines how these expressions are formatted for a print line. See the *4680 BASIC: Language Reference* for a description of a format string.

## Printing a Line of Text at the Printer

Use the WRITE FORM statement to print one line in the CR/DI or SJ station. An I/O session number, specified in the OPEN statement, indicates which printer station to use. Specify line feed information in the format string.

The printer driver prints the line asynchronously to execution of your application. You can issue multiple WRITE FORM statements to queue more than one print line to the printer driver. If you attempt to queue more than five statements, your application must wait until buffer space becomes available in the queue. Each print line must contain 38 bytes of data. The system performs line feeds after the data is printed. To line feed before printing a line, perform a WRITE FORM specifying the number of line feeds required with data of all blanks. If the data in a print line is all blanks, the system moves the print head only if necessary to return the head to home position. Home position is between the CR/DI and SJ stations.

## Controlling the Document Insert Station

Use the PUTLONG statement to control the DI station options. The following bit values refer to byte MM, which contains the DI control bits.

Bit 4 controls whether manual or automatic document insertion mode is selected. If manual mode is selected (bit 4=0), the document is inserted from the side of the DI station, manually aligned, and the station manually closed using a printer button. Keyboard input can be used to let the application know that the document is ready for printing. If automatic mode is selected (bit 4=1), the document is inserted from the front until the document is stopped by the gate. When printing is started at the DI station, the station closes and performs the print operation. The difference between manual and automatic mode as viewed from the application program is that the printer driver closes the station in automatic mode when printing is started at the DI station. In manual mode, the operator must close the station.

If a document is pulled from the top of the DI station, the station can be left closed without a document inserted. You can open the station by using the printer open and close button, or by using the CR/DI station line feed button. Your application can also issue a WRITE FORM to the CR station.

Bit 5 controls whether the application is notified if a document is removed and replaced between print lines at the DI station. If a document is not in the DI station when a print at the DI station is being performed, your application receives an error notification that the document is missing. If your application needs notification on document removal (bit 5=0), the driver remembers that the document was removed. The driver remembers this even if the document was removed while no print operation was being done. After removal, on the next print at the DI station, the application receives an error stating the document is not inserted. This error occurs even if the document was replaced before the print operation. The error notification allows your application to take action on this condition. This action could be to void the transaction or to log the occurrence. The status is reset from document-inserted to not-inserted when either the error is passed to the application or when a print is done at the CR station.

Bits 6 and 7 control options for use of a document while printing at the CR station. Documents can be inserted from the front or side of the DI station. If the document is inserted from the side, it can be aligned to reach above the print head path so that any printing at the CR station is printed on the document because it is physically in front of the CR paper. If the document is inserted from the front, it stops when the gate is reached, and printing at the CR station is done on the CR paper (not the document). If the document is inserted from the front before a DI print, the document is ready for printing without having to prompt the operator to insert the document. The printer controls the use of a document during CR printing as follows:

- If bit 6=0 and bit 7=0, a document cannot be inserted if printing is done at the CR station. A print to the CR station results in an error if a document is inserted.
- Bit 6=0 and bit 7=1 is not a valid combination. If a PUTLONG is issued with this combination, the PUTLONG results in an error.
- If bit 6=1 and bit 7=0, a document can be inserted even if a print is issued to the CR station. No error results because a document is inserted.
- If bit 6=1 and bit 7=1, a document must be placed in the DI station when printing at the CR station. If a print to the CR station is issued and a document is not inserted, an error results stating that a document is missing. This option is normally used to ensure that a document is inserted before printing on the CR receipt tape. The document can be printed after the CR receipt is printed.

When your application issues a PUTLONG statement, the options specified affect all WRITE FORM statements issued after the PUTLONG. The new PUTLONG options do not affect queued print operations resulting from WRITE FORM statements that were issued before the PUTLONG.

## Determining the Printer Status

Use the GETLONG function to determine printer status information. This function returns the current settings of the DI control bits set by PUTLONG and printer status information. The status information includes:

- Printer status (reset or not reset)
- Paper status for SJ station (low or paper jam or not low and no paper jam)
- Document insertion status (inserted or not inserted)
- DI station status (open or closed)
- Printer head status (at home or not home)
- Printer cover status (open or closed)

## Printing Checks

Checks are inserted vertically and the characters printed sideways. Check printing can be done only at the DI station of a Model 2 printer. The check must conform to the United Kingdom Association for Payment Clearing Services (APACS) standard, or all of the data to be printed on the check must fit into the area

between the 84th print position to the right of the left margin and the 156th print position to the left of the right margin. (A total of 380 print positions are available.) To print an APACS check, the check is inserted with the amount field at the top.

Check printing is useful because the customer does not have to write out the check. A blank check is handed to the cashier by the customer. The cashier inserts the check into the document insert station, and the printer types the date, amount, and payee. Then the customer signs the check.

## Formatting the Data

Your application can detect if a Model 2 printer is attached by checking bit 3 of GETLONG byte SS. If bit 3 is 1, the printer supports check printing. To use check printing, the application issues a PUTLONG to any of the printer stations with bit 3 of byte MM set to one. This instructs the printer driver to interpret the next WRITE LOGO to the DI station as a check printing command. The WRITE LOGO statement is used to print checks.

The format of the data array specified on the WRITE LOGO statement is similar to normal WRITE LOGO data. The printing field is variable in size and smaller than the 300 dot print field of a normal WRITE LOGO statement. The first three array elements contain some control information. The application can give more than one line of check data to the driver on one WRITE LOGO. This function allows the printer driver to print the check faster. The maximum amount of check data must fit into the 381 array elements used by the WRITE LOGO statement. The check data passed to the driver with the WRITE LOGO statement is in slice form, which means that each byte represents a column of eight dots on the paper. The data on a normal WRITE LOGO statement is also in slice form. The application is responsible for translation of character data to slice data and the rotation of the data for vertical printing. The system sends the data to the printer horizontally.

The printer has a total of 380 print positions: there are 190 called *primary* and 190 called *secondary*. The even-numbered print positions are primary beginning at zero, and the odd-numbered positions are secondary. The first byte in the buffer is the number of primary print positions between the home position and the first right position of the data to be printed.

**Note:** When counting print positions, begin counting from right to left. The lowest number is the print position closest to home. The highest number is the print position closest to the margin.

Check data can be printed only from primary position 78 to primary position 148. Use caution in changing the value of this byte. If one WRITE LOGO contains a different value in this byte than the previous WRITE LOGO for the same check, the printer moves the print head to home before printing the new data with the new offset. Check printing can slow if small changes are frequently made to this byte. Performance is improved if the application does not revert to the previous offset. However, increasing the value in this byte decreases the area where the print head is moved, which can improve performance.

The following control information must be contained in the first three array elements (bytes 1, 2, and 3):

- Byte 1 plus the dividend of byte 2 divided by 2 equals the number of primary print positions taken up by the print line. Byte 1 divided by 2 is less than or equal to 148.
- Byte 2 in the buffer equals the length of the data (that is, the number of bytes of data for one line of the check data). Byte 2 cannot be less than 50 bytes and should be an even integer.
- Byte 3 equals the number of lines buffered in the WRITE LOGO buffer. The buffer that passes to the printer driver on a WRITE LOGO always has 380 bytes for print data. One line of check data is always less than 377 bytes. Byte 3 should always be less than or equal to 377 when it is divided by the value of byte 2 and rounded to the nearest integer.

**Note:** To reduce processing time, the application can pass multiple check print lines to the driver simultaneously.

## Character Sets

Character sets are used to print the check data. Byte 381 contains the number of dots to advance the paper after each line is printed. If the 5 x 8 dot character set is being used, the byte should be set to 6. If the 8 x 8 dot character set is being used, the byte should be set to 9. Characters cannot be split across print lines. You should move the print head to the home position after the check is printed to ensure that the check printed correctly. There are two ways to code the application to move the print head:

- Perform a TCLOSE after the last line of the check is printed.
- On the last WRITE LOGO, set the high-order bit of byte 381 to ON. The driver looks at the high-order bit of byte 381. On the last printed line of the WRITE LOGO, the driver commands the printer to return the print head to the home position.

This procedure is faster than a TCLOSE because a TCLOSE forces the application to wait until all queued prints have completed the procedure.

## Problem Determination

To perform problem determination, an ASYNC error with the error number 80900524 results from one of the following conditions:

- The first three bytes of the data in the WRITE LOGO buffer do not conform.
- There are adjacent wire-fires in the print data.
- The printer does not support check printing.

Your application can determine the cause of the error by checking bit 3 of byte SS. The return code is ERRN=80900524H.

Your application must verify that no adjacent wire-fires are in check data. For example, if bytes 34 and 35 are part of the same print pass, ANDing the two should give you zero.

**Attention:** Adjacent wire-fires can result in the destruction of the print head. If the design of the character-to-slice translation table is correct, the application should not have to check for adjacent wire-fires.

## Programming Considerations

The following procedures are not checked by the printer driver. They should be performed by the application.

- After any document insert line feed commands, a blank APACS print should be done before printing the check. Otherwise line feeding is incomplete after the first print pass.
- The value in the low-order four bits of byte 381 of the logo buffer should not be greater than 9.
- One pass of the print head must be used to print one, and only one, column of characters. If columns of characters are split between print passes, print quality is not readable.

## Example

See Appendix C, "Character Sets and Check Printing Application," on page 741 for an example application using check printing.

## Logo Printing

Logo printing can be done only at the CR/DI station. Use the WRITE LOGO statement for printing logos.

The WRITE LOGO statement prints all-points-addressable data at the CR/DI station. The dots to be printed are specified in an array with this statement. The number of elements in the array must be a multiple of 381. Each set of 381 bytes has 380 bytes of logo print data followed by one byte that contains the number of dots that the paper is advanced after printing. Only the first 300 bytes of the 380 bytes of print data can be printed. These bytes are printed in the center 300-dot columns of the line. The system ignores the last 80 bytes of print data. Each byte controls the printing of one 8-dot column. The first byte in the array corresponds to the leftmost print position. Bit zero of each byte corresponds to the topmost dot of each dot column. If a bit=1, the corresponding dot is printed.

Only one WRITE LOGO statement can be queued. If a WRITE LOGO is issued before the previous logo print ends, execution is suspended until the previous logo print is complete. The system passes errors to the ON ASYNC ERROR subprogram. The error can be bypassed or the entire logo can be reprinted. Requests for retries of logo prints should not be made.

### Determining When Printing is Complete

If you have print lines queued and want them printed before continuing application execution, use the TCLOSE statement. This statement causes your application to suspend execution until all outstanding print lines have been printed at all printer stations or until an error is detected. If an error is detected during the completion of any queued prints, the system gives control to the ON ASYNC ERROR subprogram. When the TCLOSE is complete, the print queue is empty and the print head returns to the home position.

### Performance Considerations

There is a margin to the left of the CR/DI station and a margin to the right of the SJ station. When printing is done at a station, the print head first moves from the home position to the station margin or from the margin to the home position depending on where the head was left from the previous print. If the print head is in the margin of one station and the next print request is for the other station, the print head must move back to the home position before it reaches the station at which the print is requested. In such a case the print head travels across one station without printing.

To maximize performance, your application can print so that print head travel time is minimized. Your application should start with the head in the home position (following TCLOSE) and print an even number of times at one station before printing at the other station. In some cases your application prints the same data at both the CR/DI station and the SJ station. In such cases, you should alternate which station is printed at first (CR, SJ, SJ, CR, CR, SJ, and so on). This results in an even number of prints at one station before moving to the other.

For logo printing, the print head must travel across the print line either two or four times. Only two passes are required if you compose the logo by using every other column of dots. You should use either all odd dot columns or all even dot columns.

### Example

This example contains code for operating a printer. The program writes messages to the display, sets up the synchronous and asynchronous error handlers, and prints lines at the CR and DI stations.

```
%ENVIRON T
! Declare integers.
INTEGER*2 DISPLAY,NUMA
INTEGER*4 REQ%,STAT%
! Async error handling subprogram.
SUB ASYNCSUB(RFLAG,OVER)
INTEGER*2 RFLAG
STRING OVER
! Set up synchronous error handler
! for async subprogram.
ON ERROR GOTO SERROR
! See the error recovery example
! for the ON ASYNC ERROR CALL
! statement in the Toshiba 4680 BASIC
! Language Reference.
EXIT SUB
END SUB

Start execution.
! The alphanumeric display is opened for display of prompt.
DISPLAY = 4
OPEN "ANDISPLAY:" AS DISPLAY
CLEARS DISPLAY

WRITE #DISPLAY; "PRINTER SAMPLE"
WAIT;2000
! Branch around called routines.
GOTO START.OF.PRINTS
```

```

! Open the print station designated in the variables.
FUNCTION PRINTER
  OPEN STATIONS$ AS NUMA
  CLEARS DISPLAY
  WRITE #DISPLAY; "OPEN PRINTER ", STATIONS$
  EXIT FUNCTION
END FUNCTION
! TCLOSEs to clear all the print requests,
! then display and close the print station.
FUNCTION CLOSEA
  TCLOSE NUMA
  CLEARS DISPLAY
  WRITE #4; "CLOSE = ", NUMA
  CLOSE NUMA
  EXIT FUNCTION
END FUNCTION
! Start printing.
START.OF.PRINTS:
! Set up the asynchronous error handler
! and the synchronous error handler.
ON ASYNC ERROR CALL ASYNCSUB
ON ERROR GOTO ERR.HNDLR
! Print and space at the customer receipt station.
CR.PRINTER:
  NUMA = 3
  STATION$ = "CR:"
  CALL PRINTER
  WRITE FORM "C38 A1";#NUMA; "PRINT CASH RECEIPT AND SPACE 1 AFTER "
  WRITE FORM "C38 A2";#NUMA; "PRINT CASH RECEIPT AND SPACE 2 AFTER "
  WRITE FORM "C38 A3";#NUMA; "PRINT CASH RECEIPT AND SPACE 3 AFTER "
  WRITE FORM "C38 A4";#NUMA; "PRINT CASH RECEIPT AND SPACE 4 AFTER "
  WRITE FORM "C38 A5";#NUMA; "PRINT CASH RECEIPT AND SPACE 5 AFTER "
  CALL CLOSEA
! Print and space at the document insert station.
DI.PRINTER:
  NUMA = 2
  STATIONS$ = "DI:"
  CALL PRINTER
REQ% = 00000H
! Set manual document insert.
! Requires operator to press Close button.
PUTLONG NUMA , REQ%
INSERT.DOC:
  CLEARS DISPLAY
  WRITE #DISPLAY; "INSERT DOCUMENT"
! Test for document present.
REQ% = GETLONG (NUMA)
STAT% = REQ% AND 00000030H
IF STAT% <> 00000010H THEN GO TO INSERT.DOC
  WRITE FORM "C38 A1";#NUMA; "PRINT ON DOCUMENT AND SPACE 1 AFTER "
  WRITE FORM "C38 A2";#NUMA; "PRINT ON DOCUMENT AND SPACE 2 AFTER "
  WRITE FORM "C38 A3";#NUMA; "PRINT ON DOCUMENT AND SPACE 3 AFTER "
  WRITE FORM "C38 A4";#NUMA; "PRINT ON DOCUMENT AND SPACE 5 AFTER "
  CALL CLOSEA
  CLEARS DISPLAY
  WRITE #DISPLAY; "END OF PRINT SAMPLE"
  CLOSE DISPLAY
  GOTO END.PROG
ERR.HNDLR:
! Prevent recursion.
ON ERROR GOTO END.PROG
! Determine error type and perform appropriate
! recovery and resume steps.
END.PROG:
STOP
END

```



---

## Printer Driver Model 3 or 4/4A

This section describes the model 3 and 4/4A printer drivers and provides guidelines for using them.

### Characteristics

The Model 3 or 4/4A printer is a two-station impact printer that provides three functions. It provides a CR function in one station, an SJ function in a second station, and a DI function in the same CR and SJ stations. The CR function provides a hard copy of the transaction. It also can function as a general output device to provide data to the operator. The SJ function records data on every transaction for an audit trail or performs any other printing function that the user program dictates. The DI function allows insertion of a form, document, or check directly into the printer from the front or from the top. Use this function to print a variety of forms such as store charge account forms, or to print or endorse checks. The printer mechanism is a 9-wire dot matrix, bidirectional, all-points-addressable, high-speed device. The number 9 print wire can be used for an underscore or descenders.

The CR and SJ stations each print up to 38 characters at 15 characters per inch (CPI). The DI station prints up to 86 characters at 15 CPI. Other fonts are stored in the printer that include 12 CPI, and 7.5 CPI. The 7.5 CPI characters also can be printed double high. Lowercase characters can be printed in addition to normal characters. Double strike is available in all stations for emphasis, but at a reduced rate because of unidirectional printing.

The default line spacing for the CR, SJ, and DI stations is six lines per inch. The line spacing for each station can be changed by the PUTLONG statement and the current line spacing is returned by the GETLONG function. See the *4680 BASIC: Language Reference* for additional information on the PUTLONG and GETLONG functions.

The application program can specify how the printer driver is to interpret the line feed specification (A value) of the WRITE statement. The application can advance the paper in the various stations in increments of full lines or motor steps. (There are 9 motor steps per line at a line spacing of 8 lines per inch and there are 12 motor steps per line at a line spacing of 6 lines per inch.) If PUTLONG and GETLONG bit 4 of byte LL is zero, the driver interprets the A value of the WRITE statement as a full line feed. If bit 4 of byte LL is 1, the driver interprets the A value in motor steps. For compatibility with current applications, the driver defaults to interpreting the WRITE A value in full line feeds.

Logo printing is supported in the printer code, but at approximately half the speed because of unidirectional printing. Double high fonts are also printed at a lower speed. (Single-line logos print at full speed.) However, the driver code does not support logo printing in the SJ station. See “Logo Printing” on page 114 for additional information.

While printing in the normal mode, the print speed is up to 250 lines per minute, depending on the application.

### Compatibility with Applications Written for the Model 1 and 2 Printers

The Model 3 or 4/4A printer is compatible with applications written for the Model 1 and 2 printers, but some minor changes to existing applications will be necessary due to printer hardware differences. The following list identifies the application changes required for compatibility with the Model 3 or 4/4A printer:

- The number of line spaces to advance the paper at the end of a transaction must be increased from 8 to 10 lines. See “Receipt Paper Cutter” on page 114 for more information.
- A receipt paper cutter command must be issued after advancing the paper at the end of a transaction. See “Receipt Paper Cutter” on page 114 for more information.
- Two new printer driver return codes must be added to the application asynchronous error routine. Refer to “Journal Buffering When Printing on Documents” on page 111, “Preventing Unnecessary Reprints” on page 113, and also see the *4690 OS: Messages Guide* for more information.



Depending on how the application program has been designed to position documents, the following changes might also be required:

- The number of line spaces a document is advanced after being inserted in the document station might need to be changed. See “Manual or Automatic Document Insertion” on page 110 for more information.
- A document eject command might be required to remove a document from the document station. See “Document Eject Command” on page 112 for more information.
- It is recommended that the print head be moved to the center home position before inserting a document. If the application was not originally designed to do so, this change might be required. See “Positioning the Print Head” on page 113 for more information.

If a single application program is used with both the Model 3 and 4/4A printers and the existing Model 1 and Model 2 printers, the application program needs to determine which printer is being used and perform the preceding changes only for the Model 3 or 4/4A printer. See “Determining the Printer Status” on page 113 for more information.

If you are not familiar with 4680 BASIC or the methods used to interface to these printers, see the *4680 BASIC: Language Reference*.

## Functions Your Application Performs

Your application program can perform the following functions with the printer:

- Write characters to any station using four different character fonts
- Write characters to any station using emphasized print
- Write all-points-addressable data to the CR and DI stations
- Sense documents and control document positioning
- Change line spacing in all stations
- Perform a cut of the receipt station paper
- Obtain the printer status and printer type

## Accessing the Printer

Use the OPEN statement to gain access to the printer device driver. The name for each print station is:

**CR:** Customer receipt station  
**SJ:** Summary journal station  
**DI:** Document insert station

**Note:** The colon is part of the name, and you must open each station before it will print.

Use the CLOSE statement to end communication with a printer station. You must issue a CLOSE statement for each print station to which your application has issued an OPEN.

## Preparing a Line to Print

You can specify the data to be printed on a line in one or more expressions. Each expression can be a string or of numeric type. The format string on the WRITE FORM statement defines how these expressions are formatted for a print line. See the *4680 BASIC: Language Reference* for a description of a format string.

## Printing a Line of Text at the Printer

Use the WRITE FORM statement to print one line in the CR, DI, or SJ stations. An I/O session number, specified in the OPEN statement, indicates which printer station to use. Specify line feed information in the format string.

The printer driver prints the line asynchronously to execution of your application. You can issue multiple WRITE FORM statements to queue more than one print line to the printer driver. When the printer driver queue becomes full, your application must wait until buffer space becomes available in the queue. The

system performs line feeds after the data is printed. To line feed before printing a line, issue a WRITE FORM specifying the number of line feeds required with data of all blanks. If the data in a print line is all blanks, the system moves the print head only if necessary to return the head to a home position. The home position is between the CR and SJ stations and to the left of the CR station.

## Emphasized Printing

Emphasized, or double strike printing, is available in all printer stations at a speed that is one-third the normal printing speed. In emphasized print mode, the line is printed once, the print head is returned to the starting position, and the line is printed a second time. Therefore, every line of print requires three passes of the print head.

Emphasized printing is primarily intended to be used when printing on thick multiple-part forms. A second strike of the print head wires on most forms results in the copies having darker, more easily readable print. An application program puts the printer in emphasized mode by imbedding control characters at the beginning of the print data sent by the WRITE statement. Table 11 on page 108 shows the character sequences for emphasized printing. Emphasized mode must be used for the entire line and remains in effect for only one line. Emphasized printing is available with all printer fonts.

Table 11. Model 3 or 4/4A Printer—Emphasized Print Definition Characters

Description	Control Character Designator	Comments
Start Emphasized Print	CHR\$(27), CHR\$(69)	Results in one-third print speed

## Emphasized Printing Programming Example

The following example illustrates the method used to print a line using emphasized printing:

```

OPEN "CR:" as 5                ! REM Open Printer Receipt Station
ESTART$ = CHR$(27) + CHR$(69) ! REM Start Flag - Emphasized Printing

LINE$ = ESTART$ + " Welcome to BROOKS Department Store "
WRITE FORM "C40 A2"; #5; LINE$ ! REM Print Welcome line on Receipt
                                ! REM and space 2 lines.
WRITE FORM "C38 A1"; #5; "  January 29, 1995          10:10 a.m. "
```

### Output of Example Program

Welcome to BROOKS Department Store	
January 29, 1995	10:10 a.m.

## Font Specification

The pitch of the default font for the Model 3 or 4/4A printer is 15 CPI. You can specify a font change by imbedding a control character, followed by a character designating the font type wanted, in the data field sent to the printer by the WRITE statement (for example, CHR\$(27), CHR\$(58) results in a 12 CPI font.)

After receiving the character sequence denoting a font change, the printer continues to print in that font until the end of the line is reached or until another font change character sequence is found. At the end of the line, the printer resets to the default font of 15 CPI.

Only 15 CPI and 7.5 CPI fonts can be mixed in one WRITE statement. However, single high and double high characters cannot be mixed in one WRITE statement. If different pitch characters must be printed on the same line, the application can print the line using two WRITE statements with no line feed between the two statements. This technique requires two passes to print one line.

The application program should ensure that the number of characters specified in a WRITE statement for a given font can be printed on the station being used. The printer driver returns an illegal data error (ERR=ID; ERRN = X'80900524') if the line width exceeds the width of the station.

Table 12 on page 109 shows the maximum number of printable characters for each font type and station type. The table also shows the fonts supported by the Model 3 or 4/4A printer and the control characters used to designate the fonts. Each font control character pair inserted in the WRITE data field increases the size of the field by two bytes. To account for the larger data field size, the 4680 BASIC runtime subroutines support data fields greater than 38 characters. However, the Model 1 and Model 2 printer drivers return an error if more than 38 characters are used.

Table 12. Model 3 or 4/4A Printer Font Definition Characters

Font Description	Control Character Designator	Maximum Characters Per Line	Comments
15 CPI	CHR\$(27), CHR\$(59)	38 CR, SJ; 86 DI	Default Font
12 CPI	CHR\$(27), CHR\$(58)	30 CR, SJ; 68 DI	
7.5 CPI	CHR\$(27), CHR\$(14)	19 CR, SJ; 43 DI	15 CPI Double Wide Font
7.5 CPI Double High	CHR\$(27), CHR\$(23)	19 CR, SJ; 43 DI	Double High, Double Wide

## Font Change Programming Example

The following example illustrates the method used to change fonts with the Model 3 or 4/4A printer:

```

OPEN "CR:" as 5                ! REM Open Printer Receipt Station
FONT15$ = CHR$(27) + CHR$(59) ! REM 15 char/inch Control String
FONT75$ = CHR$(27) + CHR$(14) ! REM 7.5 char/inch Control String

LINE$ = " Welcome to " + FONT75$ + "BROOKS" + FONT15$ + " Dept. Store "
WRITE FORM "C36 A1"; #5; LINE$ ! REM Print Welcome line on Receipt

```

### Output of Example Program

```
| Welcome to B R O O K S Dept. Store |
```

In the preceding example, the 7.5 CPI pitch characters used for the word BROOKS take twice the space of characters printed at 15 CPI (the highlighted characters in the example are used to simulate a double-wide font). This extra width results in only 32 data characters filling the 38 character-wide receipt paper. It is important when mixing fonts in a single line of print that the line width of the individual printer station not be exceeded.

## Controlling the Document Insert Station

The document station on the Model 3 or 4/4A printer covers both the CR and SJ stations (on the Model 2 printer, only the CR station is blocked when a document is being printed). The document sensors are positioned at the far right of the DI station, so the document must be aligned to a fixed guide located at the right edge of the SJ station. For additional information about positioning forms, see the *4683/4684 Point of Sale Terminal: Operations Guide*, the *4680 Store System: Preparing Your Site*, and the *Store Systems: Installation and Operation Guide for Point-of-Sale Input/Output Devices*.

**Note:** The fixed guide is provided for alignment similar to the Model 1 and 2 printers. For wider documents, the document is inserted over the top of the guide and aligned against the right wall of the DI station.

When your application issues a PUTLONG statement, the options specified affect all WRITE FORM statements issued after the PUTLONG. The new PUTLONG options do not affect queued print operations resulting from WRITE FORM statements that were issued before the PUTLONG. See the *4680 BASIC: Language Reference* for additional information on the PUTLONG statement.

## Top or Front Loaded Documents

Documents can be inserted from the top or the front on the Model 3 or 4/4A/4A printer. A sensor is activated as the operator inserts the document to the positive stop feed rollers and a light emitting diode (LED) on the front of the printer lights when the sensor is activated.

An application can use the GETLONG function to determine the presence of a document. Bit 0 of byte SS is 1 when a document has been inserted in the top document station or the top of the form has been detected for a document inserted in the front document station. Bit 4 of byte SS is 1 when a document has been inserted in the front document station or the bottom of form has been detected for a document loaded in the top document station. Both top and bottom document sensors must be activated before printing can begin at the document station.

An application can use the two document sensors to sense how a document was inserted. Based on this determination, the application can order the print lines accordingly, for example, either printing the lines in reverse order for top-inserted forms or standard order for forms inserted from the front.

## Manual or Automatic Document Insertion

GETLONG or PUTLONG byte MM, bit 4, controls whether manual or automatic document insertion mode is used. If manual mode is selected, the document is inserted to the feed rollers, and the station is activated using the document station ready button on the front of the printer. Pressing the printer document station ready button causes the document to be fed into the printer until the first printable position is in the print field. If the operator wants to start printing in a position other than the top of form, the operator can use the printer line feed buttons to further position the document, or the application can advance the document.

If automatic mode is selected (bit 4=1), the document is again inserted to the feed rollers, but the station is automatically activated and the document is automatically positioned to the first print position when the first WRITE statement is sent to the document station. The application can advance the document additional spaces before printing the first line by issuing a WRITE statement with the appropriate line feed information. The difference between manual and automatic mode as viewed from the application program is that the printer driver gets the station ready in automatic mode when printing is started at the document station. In manual mode, the operator must get the station ready by pressing the document station ready button.

Documents inserted from the top block the CR and SJ stations when the form is stopped against the feed rollers. The operator must not insert a form from the top until printing at the CR station has completed. An error is returned if a WRITE statement is issued to the CR station when a document is in place.

Documents inserted from the front will not block the CR or SJ stations until manually fed into the station using the document ready button or automatically from a WRITE statement to the document station.

Since users can insert documents from either the top or the front of the Model 3 or 4/4A printer, designing an application to accommodate both insertion methods is not trivial. Mistakes could be made even if a message is displayed prompting you on the method of insertion. To compensate for user error and to make the process of designing applications easier for the programmer, the printer is designed to automatically interpret the intended insertion direction and automatically correct for user errors.

The document will be automatically registered at the top-of-form if an application program is designed for automatic front insertion of documents. The following is an example of an application program that is designed for automatic front insertion of documents:

- The document motor direction is front-to-top (bit 1 of PUTLONG byte MM is zero).
- The automatic insert bit is 1 (bit 4 of byte MM).
- The document is inserted from the front.

If, however, you insert the document from the top, the printer automatically advances the document through the printer until the document is registered at the top-of-form. The advantage to this method is that even though the application was designed for front insertion, the result of inserting the document from the top is exactly the same as inserting from the front.

The operation of the opposite insertion method is also automatic. If an application is designed for automatic top document insertion and you place the document in the front opening, the printer automatically advances the form through the printer, registering the document at the bottom-of-form.

This feature of the printer should allow for more efficient application programs and improved usability.

## Removing and Replacing a Document

Bit 5 of byte MM controls whether the application is notified if a document is removed and replaced between print lines at the DI station. If a document is not in the DI station when a print at the DI station is being performed, your application receives an error notification that the document is missing. If your application needs notification on document removal (bit 5=0), the driver remembers that the document was removed. The driver remembers this even if the document was removed while no print operation was being done. After removal, on the next print at the DI station, the application receives an error stating the document is not inserted. This error occurs even if the document was replaced before the print operation. The error notification allows your application to take action on this condition. Such action could be to void the transaction or to log the occurrence. The status is reset from document-inserted to not-inserted when either the error is passed to the application or when a print occurs at the CR station.

**Document Options:** Bits 6 and 7 control options for use of a document while printing at the CR station. These options can be used only with documents inserted from the front of the DI station. If the document is inserted from the front, it stops when the rollers are reached, and printing at the CR station is done on the CR paper (not the document). If the document is inserted from the front before a DI print, the document is ready for printing without having to prompt the operator to insert the document. The printer controls the use of a document during CR printing as follows:

- If bit 6=0 and bit 7=0, a document cannot be inserted if printing is done at the CR station. A print to the CR station results in an error if a document is inserted.
- Bit 6=0 and bit 7=1 is not a valid combination. If a PUTLONG is issued with this combination, the PUTLONG results in an error.
- If bit 6=1 and bit 7=0, a document can be inserted even if a print is issued to the CR station. No error results because a document is inserted.
- If bit 6=1 and bit 7=1, a document must be placed in the DI station when printing at the CR station. If a print to the CR station is issued and a document is not inserted, an error results stating that a document is missing. This option is normally used to ensure that a document is inserted before printing on the CR receipt tape. The document can be printed after the CR receipt is printed.

## Journal Buffering When Printing on Documents

The SJ station of the Model 3 or 4/4A printer is blocked when a document is being printed. Therefore, simultaneous printing on the DI and SJ stations is not possible. To minimize the impact to the application program, the printer driver buffers all data written to the SJ station when a document is being printed.

**Note:** Buffering of the journal data is not required when printing on the CR.

The default method of journal buffering is transparent to the application and is activated from the document sensors. The driver buffers all data sent to the SJ station after a form has been detected by the top document sensor and prints the data following the removal of the document.

The application program can control when the buffered journal data is printed by setting bit 2 of byte LL in the PUTLONG statement. When this bit is zero, the buffered journal data is printed after the document has been removed. If bit 2 is 1, the buffered data is held until this bit is reset to zero. This method of controlling the print execution of the buffered journal data can be advantageous when a transaction requires more than one document (for example, the driver continues to buffer the data until all of the documents have been printed). Immediately after resetting bit 2 from 1 to zero, the printer driver begins printing the journal data. Therefore, the application should ensure that the user has completely removed the document from the printer before resetting this bit to zero.

Use terminal configuration to specify the maximum number of journal lines the driver buffers. The driver reads the configuration data and allocates the memory required to store the number of lines specified. The driver holds this memory in reserve at all times to ensure it is available when required.

**Note:** For planning purposes, 64 bytes of memory are reserved for each line specified in configuration. This value includes 38 bytes for data and 26 bytes for line feed and other control information.

If more lines are sent to the SJ station than specified during configuration, the driver logs error message W354. The application program receives a return code, ERRN=8090052F ERR=BO, in response to WRITE statements to the SJ station after the journal buffer is exceeded. If the application receives this return code, it should eject the document and display a message indicating the journal buffer was exceeded. When the document is ejected, the data is printed (assuming bit 2 of byte LL is zero), and the journal buffer is cleared. An application program can issue a GETLONG command to determine the status of the journal buffer. If bit 0 of byte LL is zero, the journal buffer is empty. After clearing the buffer, a message can be displayed prompting the operator to reinsert the document and complete the transaction. No data is lost if the document is ejected when this error is first received. Retries are not allowed in response to the journal buffer overflow return code.

As a visual indicator that the buffer was exceeded, a line of asterisks with an error message number is also printed on the journal station following the printed data. This message alerts store personnel that an error occurred and that the journal buffer size must be increased using the configuration utility. Figure 7 on page 112 is an example of how the journal station appears after the buffer has been exceeded.

```

      1  LAYAWAY      0182  0001 110
LAYAWAY OPENED ON      2/05/95
ACCOUNT 5384
1          MDS 1      25.00
8          MDS 1      50.00
5          MDS 1      30.00
4          MDS 2       5.00
***** W354 *****
```

Figure 7. Exceeding the Journal Buffer Example

## Reinserting Documents for Printing

Some transactions require that a document be inserted into the printer many times for printing at a different location each time (for example, documenting exception conditions or for voiding transactions). These documents can be positioned for printing using the automatic method described in “Manual or Automatic Document Insertion” on page 110, or they can be positioned using a manual method for the Model 3 or 4/4A printer. The following steps are required for reinserting a document using the manual method:

1. Insert the document to the feed rollers and press the printer line advance button until the print location is positioned just above the printer cover.
2. Press the document station ready button to reverse feed the document into the printer. The document is now correctly positioned for printing and the application can issue WRITE statements to the DI station.

This feature makes the Model 3 or 4/4A printer compatible with existing application programs that require documents to be inserted from the side and manually positioned for printing.

## Document Eject Command

The document station of the Model 3 or 4/4A printer holds documents tightly in place, preventing manual repositioning. This method allows for more accurate positioning when advancing a document under program control, but it also prevents an operator from pulling a document out of the printer after printing has completed.

For fast removal of documents after printing, use the document eject command. Issuing this command results in the document being fed rapidly out of the printer until it is free from the document feed rollers. The document eject command is initiated by sending the control character sequence “CHR\$(27), CHR\$(12)” in a WRITE statement to the DI station. The direction the document is advanced out of the printer is controlled by the document line feed setting specified by PUTLONG or GETLONG byte MM, bit



1. The TCLOSE command ensures that the document eject is completed before the application is able to issue another printer command, and it is recommended that this command be used after the document eject command.

## Positioning the Print Head

Before inserting documents, the print head should be moved to the center home position. Inserting documents is easier if the print head is at the center home position. To center the print head at the home position, the application should issue a TCLOSE command before inserting the document.

**Note:** Some forms might insert more easily with the print head positioned at the far left sensor position. Testing your application with the various print head locations is highly recommended.

## Reversible Document Station Motor

The default document motor direction is the same as that used to feed the receipt paper during printing (bottom-to-top). The direction of the document motor is changed by the PUTLONG statement. The current setting of the document motor is found by using the GETLONG function. See the *4680 BASIC: Language Reference* to determine the appropriate bit mapping.

## Document Station Character Line Lengths

For maximum performance, the Model 3 or 4/4A printer driver checks the length of the data sent to the document station to determine how far the print head should travel. If 38 characters or less are written to the document station, the print head travels only half the width of the station. The data is printed right-justified over the journal station. If more than 38 characters are printed, the print head travels the entire width of the document station. Using the larger character fonts, some blanks can be truncated to meet the 38-character limit. If this occurs, increase the length of the data string to greater than 38 characters to force the print head to travel the entire distance.

## Determining the Printer Status

Use the GETLONG statement to determine printer status information. This statement returns the current settings of the DI control bits set by the PUTLONG statement and other printer status information. The status information includes:

- Printer type (Model 2, Model 3, or Model 4/4A)
- Paper status for the SJ station (out-of-paper or paper jam)
- Document insertion status (present or not present, top or front insert)
- DI station status (document ready or not ready)
- Document station line feed direction (bottom-to-top or top-to-bottom)
- Line spacing mode for any station (6 lines per inch or 8 lines per inch)
- Line spacing setting (line feeds or motor steps)
- Journal buffer status (holding buffered data or not holding buffered data)
- Current LOGO setting (300 or 380 slices to be printed)
- Printer head status (at center or left home or not at center or left home)
- Printer cover status (open or closed)

See the *4680 BASIC: Language Reference* for additional information.

## Preventing Unnecessary Reprints

The following return code indicates that the printer cover of the Model 3 or 4/4A printer has been opened:

```
ERRN = 8090052D      ERR = D0
```

The printer driver issues this return code whenever the printer cover has been opened and the outstanding print line completed successfully. No retries are required by the application in response to this return code, but the application may display an informational message indicating the printer cover is open.

**Note:** If printing is in progress when the cover is opened, the line in progress completes. However, no additional printing can occur until the cover has been closed.

## Receipt Paper Cutter

A receipt paper cutter is provided on all Model 3 and 4/4A printers and the application interface provides control characters for executing a partial cut of the paper. Since the tear bar on the Model 3 or 4/4A printer is intended only for occasional use (for example, when tearing off the excess paper after paper loading), every application must be changed to use the cutter at all times.

Table 13 on page 114 shows the characters that have been defined to initiate a partial cut.

*Table 13. Model 3 or 4/4A Printer—Receipt Paper Cutter Control Characters*

Description	Control Characters	Comments
Partial Cut	CHR\$(27), CHR\$(80)	Must be only characters in WRITE statement

The cutter control characters must be included in a WRITE statement following the last line before the cut. These characters must be the only characters included in the WRITE statement. No line advance occurs as part of the cutter command, and therefore, the A value in the WRITE FORM statement is ignored on a cutter command.

The application must advance the paper at the end of the transaction so that the last line clears the cutter. Ten line feeds are needed to clear the tear bar when the receipt station line spacing is set to 6 lines per inch. Twelve line feeds are required when the line spacing is set to 8 lines per inch.

## Receipt Paper Cutter Example

The following example illustrates the method used to issue a paper cut command.

```
OPEN "CR:" as 5
WRITE FORM "C38 A10"; #5; "THIS IS LAST LINE OF THE TRANSACTION "
WRITE FORM "2C1 A0"; #5; CHR$(27), CHR$(80) ! REM Issue cut command
                                           ! REM 'A' line advance value ignored
```

## Printer Home Sensors

The Model 3 or 4/4A printer has two home sensors: one located between the CR and SJ stations, and the other located to the left of the CR station. Bit 6 of byte SS of the GETLONG indicates when the print head is at the center home sensor. This is the same for the Model 1 and Model 2 printers. The information for the left sensor of Model 3 or 4/4A printers is made available to an application using bit 3 of byte MM.

## Left Home Command

A control character sequence is available to move the print head to the left home sensor (left of the receipt station). To move the print head to the far left, issue a WRITE statement that includes a left home control character sequence. The left home character sequence is CHR\$(27), CHR\$(76) or CHR\$(27), "L". No other data characters can be included in a WRITE statement that includes the left home control characters.

## Printing Checks

Checks can be printed in normal mode (horizontally) in the document station of the Model 3 or 4/4A printer. Therefore, the Model 2 check printing feature is not supported on the Model 3 or 4/4A printer. For additional information on the Model 2 check printing feature, see "Printing Checks" on page 101.

A check printing utility is available from Toshiba for users of Toshiba application programs (for example, General Sales Application or Supermarket) and also for users not using a Toshiba-supplied application. For additional information, contact your Toshiba marketing representative.

## Logo Printing

You can print logos only at the CR and DI stations. Use the WRITE LOGO statement for printing logos.



The WRITE LOGO statement prints all-points-addressable data at the CR and DI stations. The dots to be printed are specified in an array with this statement. The number of elements in the array must be a multiple of 381. Each set of 381 bytes has 380 bytes of logo print data followed by one byte that contains the number of dots that the paper is advanced after printing. Each byte controls the printing of one 8-dot column. The first byte in the array corresponds to the leftmost print position. Bit zero of each byte corresponds to the topmost dot of each dot column. If a bit=1, the corresponding dot is printed. Due to hardware limitations, only the first 300 bytes of the 380 bytes of print data can be printed with the Model 1 and 2 printers.

No hardware restriction exists with the Model 3 or 4/4A printer to limit the width of logo printing to 300 slices. However, to maintain compatibility with current 4680 or 4690 application programs, the Model 3 or 4/4A printer drivers default to the current logo width of 300 columns. To use the full 380-column width of the receipt or document station, set bit 1 of byte LL of the PUTLONG statement to 1. Use the GETLONG statement to determine the current bit value.

There is no provision in the Model 3 or 4/4A printer drivers for logo printing across the entire width of the 86-character wide document station. Therefore, the maximum logo width in either the receipt or document station is 380 slices.

Only one WRITE LOGO statement can be queued. If a WRITE LOGO is issued before the previous logo print ends, execution is suspended until the previous logo print is complete. The system passes errors to the ON ASYNC ERROR subprogram. The error can be bypassed or the entire logo can be reprinted. Requests for retries of logo prints should not be made.

## Determining When Printing is Complete

If you have print lines queued and want them printed before continuing application execution, use the TCLOSE statement. This statement causes your application to suspend execution until all outstanding print lines have been printed at the CR or DI stations or until an error is detected. If an error is detected during the completion of any queued prints, the system gives control to the ON ASYNC ERROR subprogram. When the TCLOSE is complete, the print queue is empty and the print head returns to the home position.

## Performance Considerations

There is a margin to the left of the CR station and a margin to the right of the SJ station. When printing is done at a station, the print head first moves from the home position to the station margin or from the margin to the home position depending on where the head was left from the previous print. If the print head is in the margin of one station and the next print request is for the other station, the print head must move back to the home position before it reaches the station at which the print is requested. In such a case, the print head travels across one station without printing.

To maximize performance, your application can print so that print head travel time is minimized. Your application should start with the head in the home position (following TCLOSE) and print an even number of times at one station before printing at the other station. In some cases your application prints the same data at both the CR station and the SJ station. In such cases, you should alternate which station is printed at first (CR, SJ, SJ, CR, CR, SJ, and so on). This results in an even number of prints at one station before moving to the other.

For logo printing, the print head must travel across the print line either one or three times. Only one pass is required if you compose the logo by using every other column of dots. To maximize performance, you should use either all odd dot columns or all even dot columns.

## Read Commands

The following two sections describe the extensions to the 4680 BASIC GETLONG and PUTLONG statements for the Model 3 printer. See the *4680 BASIC: Language Reference* for more information on the other statements used to communicate with the 4683 printer drivers.

## GETLONG Function

Use the GETLONG function to get status information from the impact printer driver.

The statement format is: `i4 = GETLONG ---(number)---`

`i4` = A 4-byte integer that represents the driver status. The integer represents information in the form RRLMMSS, where RR, LL, MM, and SS each represent one of the four bytes.

`number` = The same 2-byte integer variable or constant assigned to any one of the printer stations in the OPEN statement.

**Note:** The changes made to support the Model 3 printer have been marked with an asterisk (\*).

Byte RR is reserved for system use:

**bit 0** (X'01') = Reserved — This bit is used by the 4680 BASIC runtime subroutines and varies in value. It should be ignored by all application programs.

**bit 1** = 0 Reserved

**bit 2** = 0 Reserved

**bit 3** = 0 Reserved

**bit 4** = 0 Reserved

**bit 5** = 0 Reserved

**bit 6** = 0 Reserved

**bit 7** = 0 Reserved

Byte LL represents status information:

\* **bit 0** (X'01') = Journal buffer is empty  
= 1 Journal buffer contains data to be printed

\* **bit 1** = 0 Print only first 300 slices of LOGO data  
= 1 Print full 380 slices of LOGO data

\* **bit 2** = 0 Print buffered journal data  
= 1 Hold buffered journal data

\* **bit 3** = 1 Reserved for future optional paper cutter. Value set to 1.

\* **bit 4** = 0 WRITE FORM statement line feed specification (A value) represents line feeds (default)  
= 1 WRITE FORM statement line feed specification (A value) represents motor steps

\* **bit 5** = 0 6 lines/inch line spacing mode in the customer receipt station (default)  
= 1 8 lines/inch line spacing mode in the customer receipt station

\* **bit 6** = 0 6 lines/inch line spacing mode in the summary journal station (default)  
= 1 8 lines/inch line spacing mode in the summary journal station

\* **bit 7** = 0 6 lines/inch line spacing mode in the document insert station (default)  
= 1 8 lines/inch line spacing mode in the document insert station

Byte MM represents mode information:

\* **bit 0** = 0 Model 1 or Model 2 printer installed  
= 1 Model 3 or Model 4/4A printer installed

\* **bit 1** = 0 Document line feed is in normal mode, bottom-to-top (default)  
= 1 Document line feed is in reverse mode, top-to-bottom

\* **bit 2** = 0 No MICR installed  
= 1 MICR installed

\* **bit 3** = 0 Print head not at left position sensor  
= 1 Print head at left position sensor

\* **bit 4** = 0 Manual document insert (Documents can only be inserted from the top or front.)  
= 1 Automatic document insert. Documents can be inserted from the top or front. The printer closes the document feed rollers on the document when the first print line on the document insert station is issued. The document must cover the sensor.

\* **bit 5** = 0 Document cannot be removed and replaced between print lines on the document insert station. If you remove a document between print lines, an error occurs.  
= 1 Document can be removed and replaced between print lines on the document insert station.

- \* **bit 6** = 0 Document cannot be placed in the document insert station during printing on the customer receipt station. If you insert a document while printing on the customer receipt station, an error occurs.  
= 1 Document can be placed in the document insert station during printing on the customer receipt station.
- \* **bit 7** = 0 Document sensing not required by the document insert station for printing on the customer receipt station.  
= 1 Document sensing required by the document insert station for printing on the customer receipt station. Or, the document must be sensed during customer receipt station printing.

Byte SS represents status information as follows:

- \* **bit 0** (X'01') = 0 Document not present in top document station  
= 1 Document present in top document station
- bit 1** = 0 Reserved  
= 1 Paper error at the summary journal station (check bit 2 for more information)
- bit 2** = 0 No paper error  
= 1 Paper error at the summary journal station (for example, out-of-paper or paper jammed)
- bit 3** = 0 Reserved for Model 2 printer check printing feature
- \* **bit 4** = 0 Document not present in front document station  
= 1 Document present in front document station
- bit 5** = 0 Document insert station ready  
= 1 Document insert station not ready
- bit 6** = 0 Print head not at center home position  
= 1 Print head at center home position
- bit 7** = 0 Printer cover closed  
= 1 Printer cover open

## PUTLONG Function

Use the PUTLONG function to make changes to the customer receipt, summary journal, and document insert station mode.

The statement format is: i4 = PUTLONG ---(number)---

number = the same 2-byte integer variable or constant assigned to any one of the printer stations in the OPEN statement.

i4 = a 4-byte integer that represents the driver status. The integer represents information in the form RRRRLLMM, where RR, RR, LL, and MM each represent one of the four bytes. The first two bytes are reserved for system use.

**Note:** The changes made to support the Model 3 printer have been marked with an asterisk (\*).

Byte LL represents mode information:

- bit 0** (X'01') = 0 Reserved
- \* **bit 1** = 0 Print only first 300 slices of LOGO data  
= 1 Print full 380 slices of LOGO data
- \* **bit 2** = 0 Print buffered journal data  
= 1 Hold buffered journal data
- bit 3** = Reserved
- \* **bit 4** = 0 WRITE FORM statement line feed specification (A value) represents line feeds  
= 1 WRITE FORM statement line feed specification (A value) represents motor steps
- \* **bit 5** = 0 6 lines/inch line spacing mode in the customer receipt station  
= 1 8 lines/inch line spacing mode in the customer receipt station
- \* **bit 6** = 0 6 lines/inch line spacing mode in the summary journal station  
= 1 8 lines/inch line spacing mode in the summary journal station
- \* **bit 7** = 0 6 lines/inch line spacing mode in the document insert station  
= 1 8 lines/inch line spacing mode in the document insert station

**Notes:**

1. The function provided by bit 1 is not supported by the Model 3A fiscal printer.
2. The function provided by bit 6 is not supported by the Model 3A fiscal printer.
3. The function provided by bits 5 and 7 is not supported by the Model 3A fiscal printer in fiscal mode.
4. The changes to support DBCS have been marked with two asterisks (\*\*).

Byte MM represents mode information as follows:

**bit 0** (X'01') = 0 Reserved

\* **bit 1** = 0 Document line feed is in normal mode, bottom-to-top (default)

\* **bit 1** = 1 Document line feed is in reverse mode, top-to-bottom

\*\* **bit 1**  
= 0 Reserved

**bit 2** = 0 Reserved

\* **bit 3** = 0 Reserved

**bit 4** = 0 Manual document insert

**bit 4** = 1 Automatic document insert

\*\* **bit 4**  
= 0 Reserved

**bit 5** = 0 Document cannot be removed and replaced between print lines on the DI:

**bit 5** = 1 Document can be removed and replaced between print lines on the DI:

\*\* **bit 5**  
= 0 Reserved

**bit 6** = 0 Document cannot be placed in DI: during printing on CR:

**bit 6** = 1 Document can be placed in DI: during printing on CR:

\*\* **bit 6**  
= 0 Reserved

**bit 7** = 0 Document sensing not required by DI: for printing on CR:

**bit 7** = 1 Document sensing required by DI: for printing on CR:

\*\* **bit 7**  
= 0 Reserved

**Note:** The functions provided by bits 4, 5, 6, and 7 are not supported by the Model 3A fiscal printer.

---

## Magnetic Ink Character Recognition Support for Printers Model 3 and 4/4A

The Magnetic Ink Character Recognition (MICR) feature lets you read the account number and other information from the check. You do this by placing the document into the document insert (DI) station of a 4680 model 3R printer, or a 4690 model 4R printer that is equipped with the appropriate hardware.

**Note:** For your application to access the parsing routines described in this section, you must define the logical name ADXMICRF in the system logical names as ADX\_IDT1:ADXMICRF.DAT.

### MICR Data Format

The MICR hardware returns an ASCII string of up to 65 bytes that may include the following:

0-9	ASCII 0-9	(30H - 39H)
Transit Character	ASCII T	(54H)
On Us Character	ASCII A	(41H)
Dash	ASCII -	(2DH)
Unreadable Character	ASCII ?	(3FH)
Amount Character	ASCII \$	(24H)
blank	ASCII blank	(20H)

## Using the MICR Reader

You must place checks face down in the far right of the MICR reader for the MICR line to be read properly. You must place the check in the DI station and make the DI station ready prior to the MICR read command being sent to the device (prior to the WAIT to the DI station or the READ LINE from the DI station). See the *Store Systems: Installation and Operation Guide for Point-of-Sale Input/Output Devices* for complete information about how to use the MICR reader.

### Note

You can perform printing (franking) on the back of a check when you insert it in the far right of the DI station. However, only the last 20 characters of the narrow document print command can be printed on the check while in that position.

From the time that the check is inserted in the far right position and the MICR command is sent, until the check is removed from the printer, all DI print commands must be DI narrow print with the first 18 characters buffered with blanks. The application is responsible for ensuring that this restriction is followed. No checking will be done by the printer driver.

## Determining If a MICR Is Installed

The MICR present bit is 0X04 of the GETLONG MM byte.

## Reading from the MICR

There are two ways to do a MICR read:

- Issue a READ LINE to the DI station.

```
READ # PRTDI%; LINE MICRDATA$
```

This causes the application to wait until the MICR read has completed, or an error has occurred before the application regains control.

- Issue a WAIT to the DI station. When the WAIT completes, issue a READ LINE to obtain the MICR data.

```
WAIT PRTDI%;2000 ! wait 2 for MICR or 2 seconds
```

```
! save the completed event  
SAVE.EVENT = EVENT%
```

```
! if the MICR event completed  
IF SAVE.EVENT = PRTDI% THEN BEGIN  
    READ # PRTDI%; LINE MICRDATA$  
ENDIF
```

As with any other WAIT statement, when the first event being waited on completes, all others will be canceled. For example, if a DI wait and a timer wait are initiated and the timer event completes, the DI wait will be canceled and no data will be available for a READ LINE at that time. If a READ LINE is issued when the DI event was canceled, the MICR read command will again be passed to the hardware. The application will not regain control until it completes.

After the application obtains the MICR data, it can use the optional MICR data parsing routines described in this section or can parse the data using proprietary routines. The printer driver returns the data as passed by the hardware and performs no manipulation on the data.

## Understanding MICR Errors

Any error encountered on either the WAIT for the printer or the READ LINE to the printer will trigger the ON ERROR statement currently in effect. Only errors on printer WRITE statements will trigger the ON ASYNC ERROR in effect.

Any of the defined printer driver errors may be encountered during MICR actions. Printer driver return codes can be determined by checking the first two bytes of the four byte return code for 8090nnnnH, where nnnn is the unique portion of the return code, and the 8090 signifies that the error is from the printer driver. The following table contains the most common MICR errors.

Table 14. Common MICR Errors

4 Byte Return Code	Description
80900002H	No MICR present
80900528H	No document present in DI
80900521H	Head movement error

## Understanding MICR Parsing Routines

The MICR parsing routines provided by the system provide a terminal application with the ability to extract the following fields from the MICR line read from a check:

- Transit or routing number
- Account number
- Check sequence number

The MICR function is included by adding the following line to your link input file:

```
ADXMICRK.L86[S],      (for medium memory model)
```

or

```
ADXMICRB.L86[S],      (for big memory model)
```

To use the MICR parsing function, your application must first initialize the parsing table. Use the following functions to perform the initialization:

```
FUNCTION MICRINIT EXTERNAL
INTEGER*4 MICRINIT
END FUNCTION
```

The function reads the MICR parsing table from the ADXMICRF file using session number 64. The function returns 0 if initialization was OK. If the initialization was not OK, it returns the ERRN value of any error encountered.

After initialization you can parse MICR information read from a check using the following subprogram:

```
SUB MICRPARS(RC,MICRLine$,MICRTransit$,MICRAccount$, \
MICRSequence$) EXTERNAL
INTEGER*1 RC      !* MICR PARSING  OK = -1,
                  !*              NO MATCH = 0
STRING MICRLine$  !* MICR line read from check
STRING MICRTransit$ !* Transit number extracted
STRING MICRAccount$ !* Account number extracted
STRING MICRSequence$ !* Check Sequence number
                  !* extracted
END SUB
```

---

## 4610 Printer

This section describes the 4610 printer and provides information on accessing and using it. See the *4690 OS Application Interface Specification for 4610 Printers* for detailed programming information for the 4610 printer.

### Characteristics

The 4610 printers feature CR (thermal) stations and DI (impact) stations. The CR function provides a hard copy of the transaction; the DI function allows the insertion of a form, document, or check directly into the

printer from the front or from the side. Use the DI function to print a variety of forms. The 4610 printers use a superset of the Epson printer command set. Because the printers do not have a dedicated journal station, applications written for the 4610 printer must use electronic journaling. The thermal print station can print at a rate of 20 lines per second at 6 lines per inch or 26 lines per second at 8 lines per inch.

The 4610 printers have a resident character set that supports the following code pages: 850, 865, 437, 860, and 863. There are also four user-defined character sets for thermal printing and two character sets for impact printing.

See the *4690 OS: Planning, Installation, and Configuration Guide* for more information regarding the characters sets.

There are two different modes of printing that a 4690 application can use: stream mode and honor station mode. The default mode is stream mode with honor station mode as a printing method option.

### **Stream Mode**

The 4610 printers are stream based, which means that print commands and data of varying lengths can be sent to the printer for processing. The printer driver exerts less control over the printer commands to allow the application to use all of the capabilities of the 4610 printer family. Therefore, the application controls more responsibilities. Using stream mode, you can send a single buffer containing multiple printer commands for both the CR and DI stations.

To select a station in stream mode, the application must use the Set Print Mode command. Your application must select the appropriate station or the printer goes to the currently selected station regardless of the session to which the WRITE statement was issued.

Use a variable (for example, PRT4610) to access the 4610 printer driver in stream mode, assign a printer session to and open only the DI station.

### **Honor Station Mode**

Honor Station mode places a restriction on the application that will not allow a command sent to one station to contain printer commands to switch to the other print station. In this mode, when a print line is sent to the CR station, the printer driver adds the printer command to select the CR station to the front of the incoming data and also verifies that the data does not include a command to switch to the DI station. Printing to the DI station is also processed in this manner. Using this mode allows the CBASIC ERRF function to return the appropriate session number.

The printer driver causes the appropriate station to be selected based on the session number used in the WRITE statement. The driver also verifies that the data from the application does not contain a command to switch stations and returns an error code signaling that invalid data was passed if such a command is found.

The WRITE LOGO statement is not supported on a 4610 printer. Instead, logos can be downloaded and printed, or up to 40 logos can be downloaded and stored in the printer. These logos can then be selected by number and printed. See “Logo Printing” on page 126 for additional information.

## **Compatibility with Applications Written for the Model 3 and 4/4A Printers**

Applications written for previous models of POS printers (Model 1, 2, 3, and 4/4A) are not compatible with the 4610 family of printers. The 4610 printers do not have a dedicated journal station. Therefore, applications that use the 4610 printers must use electronic journaling. Any attempt by the application to open the SJ station results in an error.



To support the variable nature of 4610 commands and to allow the application the flexibility to access the full 4610 command set, applications written to support the 4610 printer family use the WRITE # statement instead of the WRITE FORM statement.

If you are not familiar with 4680 BASIC or the methods used to interface to these printers, see the *4680 BASIC: Language Reference*.

## Writing a Common Application for Different Printers

You can use Application Services to determine what type of printer is attached before actually opening a printer station. This allows you to create a single application that may have to be run with 4610 printers or earlier printers. Use the Application Services application status function (function 4) to determine the type of printer as well as model and feature information for the 4610 printer. The application can then provide the printer commands appropriate for the attached printer.

Offset 53 through offset 57 of the status data contains information on the attached printer.

## Functions Your Application Performs

With the 4610 printer, your application program controls the appropriate data and associated formatting commands to print the data. The following list shows some of the commands your application can control.

- Write characters to CR and DI stations using four different character fonts
- Write characters to CR and DI stations using emphasized print
- Write all-points-addressable data to the CR and DI stations
- Sense documents and control document positioning
- Change line spacing in all stations
- Perform a cut of the receipt station paper
- Obtain the printer status and printer type

## Accessing the Printer

Use the OPEN statement to gain access to the printer device driver. The name for each print station is:

**CR:** Customer receipt station

**DI:** Document insert station

**Note:** The colon is part of the name, and you must open each station before it will print.

Use the CLOSE statement to end communication with a printer station. You must issue a CLOSE statement for each print station to which your application has issued an OPEN.

## Waiting for Printer

Use the WAIT statement to wait for data to become available from the printer driver. Your application can wait on data from other device drivers at the same time as waiting for data from the printer driver. See the *4680 BASIC: Language Reference* for details on waiting on data from multiple device drivers.

## Determining if Data Has Been Received

Use the EVENT% statement to determine if data has been received from the printer driver. See the *4680 BASIC: Language Reference* for details on using the EVENT% statement.

## Reading Data From the Printer

Use the READ # LINE statement to read the data from the printer driver. The data returned from this statement indicates such as the number of remaining bytes in the driver's printer data buffer and the type of data read.

Because the 4610 printers allow the application to read various types of information from the printer, the application must first issue a WRITE command to the driver. This WRITE command must contain a driver



communications command requesting that the driver get the required information from the printer. Then, the application should issue a READ command to transfer the data from the driver's buffer to the application buffer.

**Note:** After the WRITE command is issued to request the data, control does not return to the application until either the data is available to read, or an error is encountered.

Issuing a READ statement without first requesting the data via the WRITE command moves the current printer status information into the application buffer.

## Recovering From an Error

Use the RESUME statement to recover from an error associated with an OPEN, READ, GETLONG, PUTLONG, or CLOSE statement. Use the ON ASYNC ERROR CALL statement for errors associated with a WRITE # statement.

## Preparing a Line To Print

You can specify the data to be printed on a line in one or more expressions. Each expression can be a string or of numeric type.

## Printing a Line of Text at the Printer

Use the WRITE # statement to print one line in the CR or DI stations. An I/O number, specified in the OPEN statement, indicates which printer station to use. Use the WRITE # statement to contain the print information (for example, tab stops or barcode information) to be sent to the printer.

You can send multiple lines to the printer in using a single WRITE statement. To issue a line feed, perform a WRITE statement with the line feed specified as print information. The printer must have at least two print lines available for processing while the current line is printing. To allow for this, your application must batch print lines, which can be used in several ways:

- Printing groups of lines
- Printing an entire receipt at the end of the transaction
- Printing while grouping like items into single lines

The printer wraps any data that exceeds the end of a line and prints it on the next line. If no line feed is received, the printer will not print the data transmitted, but will wait for a line feed or carriage return command.

## Font Specification

The font specified depends on whether you are using thermal printing or impact printing. Any of the fonts may be printed double-wide, double-high, or both double-wide and high.

After receiving the character sequence denoting a font change, the printer continues to print in that font until another font change character sequence is found.

The font CPI depends on the character size, the inter-character spacing, and the chosen font. You can specify a font change by imbedding a control character, followed by a character designating the font type wanted, in the data field sent to the printer by the WRITE statement.

**Thermal Printing:** There are three font sizes for the resident character set.

- Font A is 10 dots (wide) by 20 dots (high), with the last 2-dot row used for descenders.
- Font B is 12 dots (wide) by 24 dots (high), with the last 2-dot row used for descenders.
- Font C is 8 dots (wide) by 16 dots (high), with the last 2-dot row used for descenders.

**Impact Printing:** There are two font sizes for the resident character set.

- Font A is 150 half-dots per inch
- Font B is 120 half-dots per inch

## Font Change Programming Example

The following example illustrates the method used to select the CR station and define the variable to use font A.

```
! Select CR: station
CR=CHR$(1BH)+CHR$(63H)+CHR$(30H)+CHR$(02H)
!Select font A
FA=CHR$(1BH)+CHR$(21H)+CHR$(00H)

! Select CR: and font A, add text, and end with linefeed
WORK = CR+FA+"This is Font A"+LF
WRITE #PRT4610; WORK
```

## Controlling the Document Insert Station

The document sensors are positioned at the far right of the DI station, so the document must be aligned to the right edge of the DI station. For additional information about positioning forms, see the *4683/4684 Point of Sale Terminal: Operations Guide*, the *4610 SureMark Point-of-Sale Printers: User's Guide*, and the *Store Systems: Installation and Operation Guide for Point-of-Sale Input/Output Devices*.

## Document Eject Command

The document station of the 4610 printer holds documents tightly in place, preventing manual repositioning. This method allows for more accurate positioning when advancing a document under program control, but it also prevents an operator from pulling a document out of the printer after printing has completed.

For fast removal of documents after printing, use the document eject command. Issuing this command results in the document being fed rapidly out of the printer until it is free from the document feed rollers. The document eject command is initiated by sending in a WRITE statement to the DI station with the control character sequence X'0C'. The eject will feed until the document has exited the lower document sensors. The TCLOSE command ensures that the document eject is completed before the application is able to issue another printer command, and it is recommended that this command be used after the document eject command.

## Document Station Character Line Lengths

Because the 4610 printer uses a superset of the Epson printer command set, the printer will wrap any data that exceeds the end of a line and will print the data on the next line. Any print data that is less than the physical end of a line should be followed by a carriage return or line feed command or the next print data received will be printed immediately following the previous data. If no line feed is received, the printer will not print the data transmitted, but will wait for a line feed or carriage return command. It may appear that the printer is hung, when actually, the hardware is waiting on a line feed command.

## Determining the Printer Status

Use the GETLONG statement to determine printer status information. This statement returns the current settings of the DI control bits set by the PUTLONG statement and other printer status information. The status information includes:

- Whether a 4610 printer is attached
- Mode enabled (stream or honor station)
- Landscape or portrait mode

See the *4680 BASIC: Language Reference* for additional information on the GETLONG statement or the *4690 OS Application Interface Specification for 4610 Printers* for information specific to the 4610 printers.

Use the READ statement to obtain detailed status information for the 4610 printer.

## Receipt Paper Cutter

A receipt paper cutter is provided on 4610 printers and the application interface provides control characters for executing a partial cut of the paper. Every application must be changed to use the cutter at all times.

Table 15 on page 125 shows the characters that have been defined to initiate a cut.

Table 15. 4610 Printer—Receipt Paper Cutter Control Characters

Description	Control Characters	Comments
Cut	CHR\$(0CH)	Feed printed lines and cut

If the DI station is selected, this command works in a similar manner as the Form Feed.

## Receipt Paper Cutter Example

The following example illustrates the method used to issue a paper cut command.

```
OPEN "CR:" as 5
WRITE #5; "THIS IS LAST LINE OF THE TRANSACTION "
WRITE #5; CHR$(0CH)          ! REM Issue advance and cut command
```

## Printer Home Sensors

Bit 1 of byte 3 of the READ indicates whether the print head is in the cash receipt right home position. Bit 2 of byte 3 indicates whether the print head is in the left home position, and Bit 3 of byte 3 indicates whether the print head is in the document right home position.

## Printing Checks

Checks can be printed in landscape mode in the document station of the 4610 printer. The 4610 printer can process the check without taking it out of the printer. Your application would use the MICR read control character set to read the account information, frank the back of the check, and use the Flip check control character set to flip the check to print the face of the check.

A check printing utility is available from Toshiba for users of Toshiba application programs (for example, General Sales Application or Supermarket) and also for users not using a Toshiba-supplied application. For additional information, contact your Toshiba marketing representative.

## Programming Example for Check Printing

The following example shows how to have your application read the MICR data from a check, obtain the data from the driver, and flip the check. The application must first issue the MICR read driver communications command to the printer with a WRITE statement, then use a READ statement to retrieve the data from the printer driver.

```
! MICR read Driver Communications command
MICR=CHR$(04H)+CHR$(1BH)+CHR$(49H)
!Flip check
FLIP=CHR$(1BH)+CHR$(35H)

! Read MICR and flip check
!

! Read MICR data
WRITE #PRT4610; MICR

! Retrieve MICR data from the driver
READ #PRT4610 LINE MICRDATA$

! Flip check
WRITE #PRT4610; FLIP
```

You can download and print logos or download and store up to 40 logos. Use the download graphic control character sequence to download the graphics. The total number of data bytes defined for all defined graphics messages must be less than 64K – 160 bytes. If the definitions go over this limit, an error message is sent and the rest of the command is ignored. This command should only be sent when the data buffer is empty.

```
!
! Set up variables (assume string variables)
!
ERASESEC = CHR$(04H)+CHR$(1BH)+CHR$(23h)      !Erase EPROM sector
DLGRAPH  = CHR$(1DH)+CHR$(2Ah)                  !Download graphic
CONTIN   = CHR$(1BH)+CHR$(2EH)                  !Continuation command
PRTGRAPH = CHR$(1DH)+CHR$(2FH)+CHR$(00H)        !Print predefined message
```

LP1\$ = CHR\$(0AAH)+CHR\$(0AAH)+CHR\$(0AAH)+CHR\$(0AAH)+CHR\$(0AAH)+CHR\$(0AAH)+ \

CHR\$(0AAH)+CHR\$(0AAH)+CHR\$(0AAH)+CHR\$(0AAH)+CHR\$(0AAH)+CHR\$(0AAH)+ \

CHR\$(0AAH)+CHR\$(0AAH)+CHR\$(0AAH)+CHR\$(0AAH)+CHR\$(0AAH)+CHR\$(0AAH)+ \

CHR\$(0AAH)+CHR\$(0AAH)+CHR\$(0AAH)+CHR\$(0AAH)+CHR\$(0AAH)+CHR\$(0AAH)

LP2\$ = CHR\$(55H)+CHR\$(55H)+CHR\$(55H)+CHR\$(55H)+CHR\$(55H)+CHR\$(55H)+\

CHR\$(55H)+CHR\$(55H)+CHR\$(55H)+CHR\$(55H)+CHR\$(55H)+CHR\$(55H)+\

CHR\$(55H)+CHR\$(55H)+CHR\$(55H)+CHR\$(55H)+CHR\$(55H)+CHR\$(55H)+\

CHR\$(55H)+CHR\$(55H)+CHR\$(55H)+CHR\$(55H)+CHR\$(55H)+CHR\$(55H)

```
!
! Erase the stored graphics sector (01H)
!
```

```
!
! Download the graphics (using the continuation command)
! as number 1. Size is 24*8 x 16*8
!
```

126 Programming Guide

```
WRITE #PRT4610; CONTIN+LOG001$  
WRITE #PRT4610; CONTIN+LOG001$  
  
!  
! Print predefined graphic number 1  
!  
  
WRITE #PRT4610; PRTGRAPH+CHR$(01H)
```

## Determining When Printing is Complete

If you have print lines queued and want them printed before continuing application execution, use the TCLOSE statement. This statement causes your application to suspend execution until all outstanding print lines have been printed at the CR or DI stations. Do not use the TCLOSE statement from an ON ASYNC ERROR CALL statement subprogram. When the TCLOSE is complete, the print queue is empty and the print head returns to the home position.

---

## 4689 Printer

This section describes the 4689 printer and provides information on accessing and using it. See the *4680 BASIC: Language Reference* for detailed programming information for the 4689 printer.

### Characteristics

The 4689 printer offers fast printing and low noise (of the thermal station). To take full advantage of the increased speed of the thermal print station, you must modify the application. The operating system and the microcode have several features that can be used to reach the speed potential of the 4689 printers.

The 4689 thermal printer must continually have data available in the printer's buffer to reach or maintain rated print speeds. The hold buffer command can be used to have the printer store data temporarily in an internal 4K buffer. The data is printed when the printer receives a release buffer command. You can use this method to save data for post printing of receipts.

The 4689 printer has a resident character set that supports two font sizes. In addition to these character fonts, the user-defined characters by 24x24 dots matrix is supported. The two font sizes are:

#### 16 dots font

- 7 dots wide by 16 dots high (SBCS, Hankaku)
- 15 dots wide by 15 dots high (DBCS, Zenkaku)

#### 24 dots font

- 24 dots wide by 24 dots high (SBCS, Hankaku)
- 24 dots wide by 24 dots high (DBCS, Zenkaku)

The 4689 printer provides two different modes of printing that a 4690 application can use. The default mode is stream mode with honor station mode as an option of printing.

### Stream Mode

The 4689 printers are stream based. This means that print commands and data of varying lengths can be sent to the printer for processing. The printer driver exerts less control over the printer commands. Therefore, the application controls more responsibilities. The maximum print buffer size is 244 bytes. An error is returned if a buffer larger than this is used.

The application must select the station using the Set Print Mode command. Although communications to the printer driver is available using both the CR and the SJ for BASIC language compatibility, this is not used by the driver to select the print station. The application must select the station in the printer data stream. The CR and SJ printer stations continue to be available. However, because the 4689 printer

commands set allows the application to select the print station in the data stream, an application may access any available printer station by writing to the session opened for the CR station.

However, it may enhance program readability to perform document prints by writing to the session opened for the SJ station. Both methods work, but the application must select the appropriate station or the printer goes to the currently selected station regardless of the session to which the WRITE statement was issued.

The recommended method for accessing the 4689 printer driver in stream mode is to use a variable, such as PRT4689, to assign a printer session to and open only the SJ station. This would also mean that using the 4689 printer requires only a single session number to be used instead of three for other printers.

## Honor Station Mode

Honor station mode allows application to be more compatible with previous methods of accessing the printer. The driver adds the correct 4689 printer command to select the print station to the data sent by the application based on the session number. Use the PUTLONG statement to select the mode. The ERRF function will contain the session number of the failing station in case of error.

The printer driver causes the appropriate station to be selected based on the session number used in the WRITE statement. The driver also verifies that the data from the application does not contain a command to switch stations and returns an error code signaling that invalid data was passed if such a command is found.

## Compatibility with Applications Written for 4610 Printers

Applications written for 4689 printers are not compatible with the 4610 printers. The 4689 printer does not have a dedicated document insert station. Therefore, any application written for the 4610 printer that attempts to open the DI station results in an error. The 4689 printer has an SJ station that an application can open.

Applications written to support the 4689 printer will use the WRITE # statement to write to the printer instead of the WRITE FORM # statement used for previous printers. This allows the flexibility of accessing the full 4689 command set.

You can use Application Services to determine what type of printer is attached before actually opening a printer station. This allows you to create a single application that may have to be run with 4689 printers or earlier printers. Use the Application Services application status function (function 4) to determine the type of printer as well as model and feature information for the 4689 printer. The application can then provide the printer commands appropriate for the attached printer.

## Accessing the Printer

Use the OPEN statement to gain access to the printer device driver. The name for each print station is:

**CR:** Customer receipt station

**SJ:** Summary journal station

**Note:** The colon is part of the name, and you must open each station before it will print.

Use the CLOSE statement to end communication with a printer station. You must issue a CLOSE statement for each print station to which your application has issued an OPEN.

## Waiting for the Printer

Use the WAIT statement to wait for data to become available from the printer driver. Your application can wait on data from other device drivers at the same time as waiting for data from the printer driver. See the *4680 BASIC: Language Reference* for details about waiting for data from multiple device drivers.

## Determining if Data Has Been Received

Use the `EVENT%` statement to determine if data has been received from the printer driver. See the *4680 BASIC: Language Reference* for details about using the `EVENT%` statement.

## Recovering From an Error

Use the `RESUME` statement to recover from an error associated with an `OPEN`, `READ`, `GETLONG`, `PUTLONG`, or `CLOSE` statement. Use the `ON ASYNC ERROR CALL` statement for errors associated with a `WRITE #` statement.

## Writing to the Printer

On 4689 printers, the `WRITE #` statement is used to provide data to the printer. The application is responsible for providing the appropriate data and associated formatting commands to print the data. The 4689 printers will wrap any data that exceeds the end of a line and print it on the next line. Any print data that is less than the physical end of a line should be followed by carriage return or line feed commands or the next print data received will be printed immediately following the previous data. If no line feed is received, the printer will not print the data transmitted, but will wait for a line feed or carriage return command. It might then appear that the printer has stopped, when actually, the printer is waiting for a line feed command.

## Returning Invalid Data

There are two return codes that indicate that the data sent to the printer on a `WRITE #` statement is incorrect.

- `X'80900524'`
- `X'80901524'`

Neither of these commands can be retried using a `RESUME` statement because the data is incorrect and cannot be processed. The `X'80900524'` error code indicates that the data contained at least one of the following types of data:

- A command not included in the documented command set
- A command that is not completed before the end of the print buffer (unless the command is a download command that allows continuations)
- If honor station mode is enabled, an attempt was made to select a print station other than that of the current session number

The `X'80901524'` error code indicates that the write data buffer is too large. The maximum write data buffer is 244 bytes.

## Determining the Printer Status

Use the `GETLONG` statement to determine printer status information. This statement returns the current settings of the DI control bits set by the `PUTLONG` statement and other printer status information. The status information includes:

- Whether a 4689 printer is attached
- Mode enabled (stream or honor station)
- Landscape or portrait mode

See the *4680 BASIC: Language Reference* for additional information about the `GETLONG` statement.

## Determining When Printing is Complete

If you have print lines queued and want them printed before continuing application execution, use the `TCLOSE` statement. This statement causes your application to suspend execution until all outstanding print lines have been printed at the CR or SJ stations. Do not use the `TCLOSE` statement from an `ON ASYNC ERROR CALL` statement subprogram. When the `TCLOSE` is complete, the print queue is empty and the print head returns to the home position.



## Line Count

To provide error recovery during buffered printing (either when lines are buffered in the printer or are actually held) the printer hardware and the printer driver keep a line count for the commands going to the printer. The line count is incremented when any of the Print Character Commands are encountered, as well as some other special commands. The line count is reset when a TCLOSE is processed. Status byte 6 contains the line count of the last print line that has successfully completed. The application has access to the current line count using the READ command to the printer. However, the printer hardware and the printer driver are responsible for coordinating the line count usage.

## 4689 Native Mode Support

The 4689 Native Mode provides a basic programming interface to the 4689 printer hardware. The full functionality of the 4689 thermal printer can be used for SBasic applications. These functions include:

- Bypassing the print stream conversion
- Treating a print request with line feed as **line feed and print**
- Supporting **buffered print**
- Adding capability to issue print requests to both the receipt station and the journal station through one unit of the printer driver
- Returning a **paper near end** warning from both the receipt station and the journal station of the 4689 printer hardware to the application

## Switching to 4689 Native Mode

The default mode of the 4689 printer is 4610 emulation mode. To switch modes between 4610 emulation mode and 4689 native mode, the application issues X'1B;55;N' where:

**N** = 0 to switch the printer to 4610 emulation mode

**N** = 1 to switch the printer to 4689 native mode

When the printer is switched to a mode, the printer retains that mode until another application performs a mode switch or the printer is powered off. When the printer is in 4689 native mode, an application must issue 4689 native mode commands.

## Determining the Printer Mode

Use the GETLONG statement to determine if the printer is in 4689 native mode or 4610 emulation mode. GETLONG returns a 4-byte value of EELMMSS. The printer is in 4689 native mode if the seventh bit of SS is on. See the *4680 BASIC: Language Reference* for details about using the GETLONG statement.

## Commands

The 4689 printers have commands to specialize and set each printer to improve the usability, performance, and uniqueness. This flexibility is provided through the use of EEPROM memory. The data stored in these memory devices stays valid until it is redefined. The 4689 printer supports both 4689 native mode and 4610 emulation mode. In 4689 native mode, applications must issue the 4689 native commands. The following commands are valid only in 4610 emulation mode.

### Logo Printing

Use the following command to store the user-defined character/logo buffer.

#### RS-485 Syntax:

X'1D;2A;logo#;data'

#### BASIC Syntax:

PSTR1\$=CHR\$(1DH)+CHR\$(2AH)+CHR\$(LOGO#)

PSTR2\$= DATA



PSTRING\$=PSTR1\$+PSTR2\$

**Where:**

**logo#** High address byte for Load UDC/Logo Buffer, where:

**X'00' to X'1F'**

Logo

**X'20' to X'32'**

UDC

**data** The data to form the UDC/Logo buffer size should be 256 bytes.

**Remarks:**

You must use the WRITE LOGO statement to use this command.

**Set Print Mode**

**RS-485 Syntax:**

X'1B;21;n'

**BASIC Syntax:**

PSTRING\$=CHR\$(1BH)+CHR\$(21H)+CHR\$(N)

**Where:**

n		Bit # ↓	Function	Bit=0	Bit=1
LSB	0	0	Reserved		
	1	1	Font	24 dots	16 dots
	2	2	Overline	Cancel	Set
	3	3	Dark Shading	Cancel	Set
	4	4	Double-High	Cancel	Set
	5	5	Double-Wide	Cancel	Set
MSB	6	6	Light Shading	Cancel	Set
	7	7	Reserved		

**Default:**

n = X'00'

**Set/Cancel Double Wide**

**RS-485 Syntax:**

X'1B;57;n'

**BASIC Syntax:**

PSTRING\$=CHR\$(1BH)+CHR\$(57H)+CHR\$(N)

**Where:**

**n**

00 — Cancel Double-Wide Mode  
non-zero — Set Double-Wide Mode

**Default:**

n = 00

**Set/Cancel Double High**

**RS-485 Syntax:**

X'1B;68;n'

**BASIC Syntax:**

PSTRING\$=CHR\$(1BH)+CHR\$(68H)+CHR\$(N)

**Where:**

n

00 — Cancel Double-High Mode

non-zero — Set Double-High Mode

**Default:**

n = 00

**Remarks:**

For better print quality with double-high characters, set the printer to unidirectional print mode.

**Set/Cancel Keisen****RS-485 Syntax:**

X'1B;2D;n'

**BASIC Syntax:**

PSTRING\$=CHR\$(1BH)+CHR\$(2DH)+CHR\$(N)

**Where:**

n

00 — Cancel keisen mode

02 — Set solid keisen line

03 — Set dashed keisen line

**Default:**

n = 00

**Remarks:**

The keisen is printed in the character area. Therefore, the character code for the keisen must be sent. The valid keisen codes are:

X'01'

X'02'

X'03'

X'04'

X'05'

X'06'

X'10'

X'15'

X'16'

X'17'

X'19'

When the keisen character code does not exist, even if the mode is set to On, the keisen is ignored.

**Set/Cancel Overline****RS-485 Syntax:**

X'1B;5F;n'

**BASIC Syntax:**

PSTRING\$=CHR\$(1BH)+CHR\$(5FH)+CHR\$(N)

**Where:**

n

- 00 — Cancel Overline Mode
- 02 — Set solid overline
- 03 — Set dashed overline

**Default:**

n = 00

**Remarks:**

This is only valid in the Cash Receipt Station.

## Set/Cancel Light Shading

**RS-485 Syntax:**

X'1B;48;n'

**BASIC Syntax:**

PSTRING\$=CHR\$(1BH)+CHR\$(48H)+CHR\$(N)

**Where:**

n

- 00 — Cancel Light Shading
- non-zero — Set Light Shading

**Default:**

n = 00

**Remarks:**

This is only valid in the Cash Receipt Station.

## Set/Cancel Dark Shading

**RS-485 Syntax:**

X'1B;47;n'

**BASIC Syntax:**

PSTRING\$=CHR\$(1BH)+CHR\$(47H)+CHR\$(N)

**Where:**

n

- 00 — Cancel Dark Shading
- non-zero — Set Dark Shading

**Default:**

n = 00

## Set Print Station

**RS-485 Syntax:**

X'1B;63;30;n'

**BASIC Syntax:**

PSTRING\$=CHR\$(1BH)+CHR\$(63H)+CHR\$(30H)+CHR\$(N)

**Where:**

<b>n</b>	Specifies the print station.	
	Bit # ↓	Station
LSB	0	Reserved
	1	Cash Receipt Station
	2	Summary Journal Station
	3	Reserved

4	Reserved
5	Reserved
6	Reserved
7	Reserved

**Default:**

n = 02

**Remarks:**

Only one station may be set at a time. If more than one station is selected the command is ignored.

## Set/Cancel Rotated Characters

**RS-485 Syntax:**

X'1B;56;n'

**BASIC Syntax:**

PSTRING\$=CHR\$(1BH)+CHR\$(56H)+CHR\$(N)

**Where:**

n

00 — Cancel Rotated Characters

01 — Set Rotated Characters

**Default:**

n = 00

**Remarks:**

Rotation is 90 degrees clockwise.

## Bar Code Commands

**RS-485 Syntax:**

X'1D;6B;n;data;00'

**BASIC Syntax:**

PSTRING\$=CHR\$(1DH)+CHR\$(6BH)+CHR\$(N)+DATA+CHR\$(00H)

**Where:**

n

Bar Code

00 — UPC-A

01 — UPC-E

02 — JAN13 (EAN)

03 — JAN8 (EAN)

04 — NW7

**data** ASCII representation of the characters to be printed.

**Remarks:**

This is valid only at the beginning of a line. Printing will not start until a X'00' is received or the end of a message to the printer (data packet). Data after any incorrect character for a particular bar code will be thrown away. The printer will continue to wait for a X'00'. Excess characters will be discarded. If a X'00' or an incorrect character is received before the required number of data bytes, zeroes will be inserted following the data until the required number of bytes is reached (valid for UPC-A, UPC-E, JAN8, AND JAN13).

## Print Character Commands

The following commands should be sent after ASCII data is sent to the printer and is being held in the print buffer. Any of these commands increment the line count by 1.

**Print and Line Feed:** Use this command to print the data in the print buffer and feed paper by a pre-set amount.

**RS-485 Syntax:**

X'0A'

**BASIC Syntax:**

PSTRING\$=CHR\$(0AH)

**Print and Line Feed:** Use this command to print the data in the print buffer and feed paper by a pre-set amount.

**RS-485 Syntax:**

X'0D'

**BASIC Syntax:**

PSTRING\$=CHR\$(0DH)

**Print, Form Feed (FF) for SJ:, LOGO and Cut the Paper for CR::** Use this command to feed the paper at 48 steps (6mm) for the SJ station, print the logo, and cut the paper for the CR station.

**RS-485 Syntax:**

X'0C'

**BASIC Syntax:**

PSTRING\$=CHR\$(0CH)

**Print and Feed Paper n Lines:** Use this command to print the data in the print buffer and feed paper by the number of lines specified in the command.

**Note:** An attempt to feed zero lines on the CR station will cause one line feed.

**RS-485 Syntax:**

X'1B;64;n'

**BASIC Syntax:**

PSTRING\$=CHR\$(1BH)+CHR\$(64H)+CHR\$(N)

**Where:**

**n** Specifies the number of line feeds.

## Print Graphics

**Image Print Command:** Use this command to print an image.

**RS-485 Syntax:**

X'1B;5E;n1;n2;data'

**BASIC Syntax:**

PSTRING\$=CHR\$(1BH)+CHR\$(5EH)+CHR\$(n1)+CHR\$(n2)

PSTR2\$=DATA

PSTRING\$=PSTR1\$+PSTR2\$

**Where:**

**n1** Low byte

**n2** High byte

**data** The data to form the graphic image.

**Print Predefined Graphics (Logo) and Cut Command:** Use this command to print a predefined all-points-addressable print message and fully cut the receipt paper.

**RS-485 Syntax:**

X'1D;2F;m;logo#'

**BASIC Syntax:**

PSTRING\$=CHR\$(1DH)+CHR\$(2FH)+CHR\$(M)+CHR\$(LOGO#)

**Where:**

**m** Reserved

**logo #** Always number 1.

**Return Home (Select Print Head Location)**

Use this command to set the cutter to the home position.

**RS-485 Syntax:**

X'1B;3C;n'

**BASIC Syntax:**

PSTRING\$=CHR\$(1BH)+CHR\$(3CH)+CHR(N)

**Where:**

**n** Print head position

**Remarks:**

This command is only valid at the beginning of the line.

**Partial Cut**

Use this command to cut the cash receipt paper.

**RS-485 Syntax:**

X'1B;6D'

**BASIC Syntax:**

PSTRING\$=CHR\$(1BH)+CHR\$(6DH)

**Data Buffer Management/Batch Printing**

**Hold Printing Until Buffer is Released:** Use this command to delay printing until the buffer is released. If the system cannot send data to the printer at a speed needed to keep the printer in constant motion, the system may hold the printer queue until it has sent all the data lines for a transaction.

**RS-485 Syntax:**

X'1B;37'

**BASIC Syntax:**

PSTRING\$=CHR\$(1BH)+CHR\$(37H)

**Remarks:**

In the Thermal print stations, the printer must be printing one line while processing the next line. If the printer is unable to completely process a line before the previous line is finished, the print speed reduces by half. The line count will be reset when the buffer is held.

The hold command is a buffered command. If you issue a hold command, then issue a second hold command before a release buffer command, the second hold command will remain in the printer buffer. When the release buffer command is issued and the second hold command is encountered, printing will again be held.

If the printer driver buffer becomes full while a hold buffer command is in effect, the driver will issue a release buffer to the printer. The printer driver buffer is 4096 bytes. The application can change the size of this buffer using the PUTLONG command.

If the application issues a TCLOSE command while a hold buffer command is in effect, the driver will issue a release buffer command before the TCLOSE is processed.

**Release Print Buffer:** Use this command to release the print buffer for printing.

**RS-485 Syntax:**

X'10;05;31'

**BASIC Syntax:**

PSTRING\$=CHR\$(10H)+CHR\$(05H)+CHR\$(31H)

**Remarks:**

If an error occurs during the transaction that was being held, the printer will send back the line number on which it occurred with the error status. The system can then decide to cancel or continue the printing when the error is corrected. The command will continue the printing after the error is fixed.

## Driver Communications Commands

The driver communications commands allow the application to issue requests to the driver. As the driver receives the request, it may, in turn, send requests to the printer.

**EC Read:** Use this command to get the EC data from the printer.

**RS-485 Syntax:**

X'04;80'

**BASIC Syntax:**

PSTR1\$=CHR\$(04H)

PSTR2\$=CHR\$(80H)

PSTRING\$=PSTR1\$+PSTR2\$

**Status Read:** Use this command to get the printer status.

**RS-485 Syntax:**

X'04;20'

**BASIC Syntax:**

PSTR1\$=CHR\$(04H)

PSTR2\$=CHR\$(20H)

PSTRING\$=PSTR1\$+PSTR2\$

**Device Information Request:** Use this command to get the device specific information.

**RS-485 Syntax:**

X'04;01'

**BASIC Syntax:**

PSTR1\$=CHR\$(04H)

PSTR2\$=CHR\$(01H)

PSTRING\$=PSTR1\$+PSTR2\$

---

## Fiscal Printer Support

The fiscal printer performs fiscal commands in accordance with the tax laws of the particular country supported. Every fiscal command starts with X'1B66' (ESC^f in ASCII) that is unique to fiscal commands.

Applications written for the standard Model 3 printer are compatible with the Fiscal Printer Model 3A and Model 3F. Logo commands cannot be printed by the fiscal printer, but no error occurs if a logo print command is performed. See the *Fiscal Printer Software Supplement* for your specific country, for a detailed description of the fiscal commands.

## Application Programming for the Operating System Fiscal System

The command used to print lines on both the customer receipt and summary journal stations is the 4680 BASIC WRITE FORM command. This command remains unchanged for any existing print option application. Application programs issue fiscal commands using the WRITE FORM command. These commands provide the capability to release sales slips, daily closing vouchers, and fiscal documents. Fiscal vouchers are printed on the customer receipt station and then duplicated on the summary journal print station. The non-fiscal print command is used to provide a line feed or eject at the print stations.

The application opens the document insert station when issuing fiscal commands. Opening the document insert station allows 115 bytes of data to be sent with the fiscal commands. Opening the customer receipt or summary journal station only allows 75 bytes of data and may not be enough for some fiscal commands.

**Note:** The 4680 BASIC WRITE LOGO command and special characters are not supported by the 4680 Fiscal System.

### Error Handling and Recovery Procedures

For user programming errors, see the *4680 BASIC: Language Reference* for details regarding error reporting procedures and user actions. For functional hardware errors, see the *Fiscal Printer Hardware Supplement* for your specific country.

Lines reprinted by the fiscal unit (when applicable) are identified by the number sign (#).

### Reading from the Model 3 Fiscal Printer

The CBASIC language does not permit reading from the printer. This function is provided by a library routine in the file ADXFISRL.286 (for medium memory models) or ADXFISBL.286 (for big memory models), that must be linked with applications using the fiscal printer.

The function is called FISREAD and has the following definition:

```
SUB FISREAD(FISRC,FISERR,FISDATA) EXTERNAL
INTEGER*4 FISRC
INTEGER*2 FISERR
STRING FISDATA
END SUB
```

FISRC is a 4-byte return code from the printer driver.

FISERR is a 2-byte error code with the following possible values:

- 0 = no error
- 1 = error on open
- 2 = error on read

FISDATA is a string variable with the format:

- Bytes 0–3 are the first 4 bytes of printer status
- Bytes 4–5 are reserved for service personnel
- Byte 6 is a read flag
  - X'00' indicates this data has not been read by the application before. This is the first time data is read from the buffer, and the status matches the read data.
  - X'01' indicates the data has been read by the application previously. This occurs if no fiscal read commands (X'DA' or X'DB') were processed by the printer since the last FISREAD command was issued. This data has been read from the buffer previously and the status matches the read data.
  - X'02' indicates this is the first time data is read from the buffer, but the status bytes have been updated after the read data was received.



- X'03' indicates this data has been read from the buffer previously, but the status bytes have been updated after the read data was received.
- Byte 7 is the Microcode EC (engineering change) level.
- Bytes 8–9 contain the length of the fiscal read response. These two bytes are in INTEGER\*2 format.
- Bytes 10–265 contain the response to the fiscal read command.

## Using the FISREAD call

To read the printer data, the following sequence should be used:

1. Write the FISCAL READ command (X'DA' or X'DB') to the printer with a WRITE statement.
2. Issue a TCLOSE to the printer to flush the print buffer.
3. Call the FISREAD routine to retrieve the fiscal data into the application buffer.
4. Check the FISERR value for an error indication.
5. For an error, obtain the return code from the FISRC variable, or if no error, extract the needed data from the FISDATA string variable.

An example of a CBASIC FISREAD call is:

```
CALL FISREAD(FISRC,FISERR,FISDATA)
```

---

## Scale Driver

This section describes the scale driver and provides guidelines for using it.

### Characteristics

- One scale can be attached to a terminal and can be one of the following:
  - Non-IBM scale attached to Feature Expansion Card B or C (4683 only)
  - Integrated scanner/scale attached to a terminal socket

See the *4690 OS: Planning, Installation, and Configuration Guide* for the terminal sockets that support a scanner/scale on your terminal.

- You can select to have either English or Metric weight
  - English - weight returned is:
    - maximum of four digits
    - hundredths of pounds
  - Metric - weight returned is:
    - maximum of five digits
    - thousandths of kilograms

## Accessing the Scale

Use the OPEN statement to gain access to the scale driver.

Use the CLOSE statement to end communication with the scale driver.

## Receiving Data

Issue a READ FORM statement to receive data from the scale. If valid data is available, the variable specified in the READ FORM statement contains the weight. If an error occurs, control passes to the ON ERROR routine.

## Example

To run the following example, put a weight on the scale. The program reads the scale, displays the weight one time, and then stops.

```

%ENVIRON T
! Declare variables.
INTEGER*4 hx%,sx%,WEIGHT%
INTEGER*2 SCALE%,ANDSP%, THE.SUM%,s%
STRING ERRFX$, A$
ON ERROR GOTO ERRORA
! Indicate "ON ERROR" label.
ANDSP% = 1
! Set alphanumeric display session number to 1.
SCALE% = 2
! Set scale session number to 2.
OPEN "ANDISPLAY:" as ANDSP%
! Open alphanumeric display.
CLEAR$ ANDSP%
! Clear alphanumeric display.
WRITE #ANDSP% ; "SAMPLE SCALE PROG"
WAIT;5000
! Wait 5 seconds so display can be read.
CLEAR$ ANDSP%
! Clear alphanumeric display.
OPEN "SCALE:" AS SCALE%
! Open the scale.
WRITE #ANDSP% ; "SCALE OPEN"
! Indicate the scale was opened.
WAIT;5000
! Wait 5 seconds so display can be read.
CLEAR$ ANDSP%
! Clear display.

WRITE #ANDSP% ; "READ THE SCALE NOW!"
READ FORM "I4" ; #SCALE% ; WEIGHT%
WRITE FORM "C10 PIC(####)" ; #ANDSP% ; " WEIGHT = ",WEIGHT%
WAIT;6000
! Wait 6 seconds so weight can be read.
LOCATE #ANDSP% ; 2,1
! Locate to second line.
CLOSE SCALE%
! Close scale.
WRITE #ANDSP% ; "SCALE IS CLOSED"
WAIT ; 5000
! Wait 5 seconds so message can be read.
STOP
ERRORA:
! ERROR ASSEMBLY ROUTINE
hx% = ERRN
ERRFX$ = ""
for s% = 28 to 0 STEP -4
    sx% = shift(hx%,s%)
    THE.SUM% = sx% AND 000fh
IF THE.SUM% > 9 THEN \
    THE.SUM%=THE.SUM%+55 \
ELSE \
    THE.SUM%=THE.SUM%+48
    A$=CHR$(THE.SUM%)
    ERRFX$ = ERRFX$ + A$
NEXT s%
CLEAR$ ANDSP%
WRITE #ANDSP%; "ERR=",ERR,"  ERRL=",ERRL
LOCATE #ANDSP%;2,1
WRITE #ANDSP%; "ERRF=",ERRF%,"  ERRN=",ERRN$
WAIT ;15000
RESUME
END

```

---

## Serial I/O Communications Driver

This section describes the serial I/O communications driver and provides guidelines for using it.

### Characteristics

The serial I/O communications driver has the following characteristics:

- Each 4683 Point of Sale Terminal supports a maximum of two feature expansion cards. Each of these can contain two ports to attach serial communication I/O devices. One port can attach an I/O device that is compatible with an Electronics Industries Association (EIA) RS-232C interface and the other port can attach an I/O device compatible with an EIA RS-232C interface or an asynchronous 20-milliampere current-loop interface.
- Each port can be used in either a half-duplex or full-duplex mode.
- The terminal serial I/O port functions as Data Communications Equipment (DCE) and the serial communication device functions as Data Terminal Equipment (DTE) in terms of RS-232C interface.
- 4693 terminals support serial ports COM1 and COM2 on the native serial ports and COM1 through COM8 on the Dual Async adapter. 4694 terminals and 4683-4x1 terminal upgrades support serial ports COM1 through COM2 on the native serial ports. These serial I/O ports function as DTEs, and the serial communication device functions as DCEs as related to an RS-232C interface.

To attach a device to the serial I/O port, you need the interface requirements of the particular device and the EIA RS-232C or asynchronous current-loop interface requirements. Current loop is supported on 4683 terminals.

Commands allow you to communicate with the device. Your application has responsibility for any protocol requirements such as checkpointing, pacing, positive or negative response, or error recovery.

- | • TCxWave 6140 Series machines support Serial communication using an RS232-to-USB dongle.
- | Approved dongles are listed in the TCxWave Hardware Documentation. This does not change the
- | command set and status expectations for Serial I/O communication.

### Functions Your Application Performs

An application program can perform the following functions with the device attached to the serial I/O port (depending on the capability of the device) :

- Transmit data to the device
- Receive data from the device
- Set communication parameters
- Obtain status

### Accessing the Serial I/O Port

Use the OPEN SERIAL statement to gain access to the serial I/O driver. Specify the following parameters on the OPEN SERIAL statement:

- Serial I/O port to open (1 through 4).
- Transmit or receive speed (110, 300, 1200, 2400, 4800, 9600, or 19 200 bits per second).
- Parity (even, odd, or none).
- Number of data bits per character (5 through 8).

If the bits-per-character is less than eight, the significant bits are placed in the rightmost bits of each byte in the application buffer on a READ, and taken from the rightmost bits on a WRITE.

- Number of stop bits (1, 1.5, or 2).
- Maximum application buffer size (1 through 247).

You determine how to set these parameters by the interface requirements of the device attached to the serial I/O port. The maximum application buffer size is the maximum number of bytes that you will transmit or receive from the device on a single write or read operation.

Once you have issued the OPEN SERIAL to a port, you cannot change any of the parameters associated with the OPEN SERIAL unless you issue a CLOSE to delete your I/O session and then issue another OPEN SERIAL with different parameters.

Use the CLOSE statement to end your application's use of the serial I/O port.

## Overview of Receiving Data

Your application must enable receive mode on the serial I/O feature expansion to allow data to be received. You should also set an appropriate intercharacter timeout value. An intercharacter timer is started (using this value as the upper limit) each time a character is received. As data characters are received, they are placed in a 247-byte driver buffer.

Data in the driver buffer becomes available to your application when any of the following conditions occur:

- Data has been received and the timer expires before the next character is received.
- The driver buffer is full.
- Data has been received and an error or special event is detected (see “Waiting for Received Data” on page 143 for a list of the errors or events).

The normal condition that causes data to be available to your application should be the expiration of the intercharacter timer. This value must be compatible with the device sending the data so that the timer expires before 247 bytes are received. If the driver buffer is filled, data is lost if more data is received before the application reads the driver buffer data.

If your device can send multiple blocks of data without a response from your application, you should read the driver buffer promptly following the availability of data. If data becomes available because the timer expired, and new data characters are received before the application performs a read operation, the new characters are added to the driver buffer data. This could result in a buffer overrun condition if the driver buffer data is not read promptly.

If data has been received and an error is detected, the error is not reported until the application reads the data already received. The error is reported the next time the application performs a wait or read operation. Any data received following the error is discarded until the application is notified of the error.

If the driver buffer is empty and an error occurs, the application is notified of the error on the next wait or read operation. Any data received before the application is notified of the error is discarded. Note that data detected to be in error is never passed to the application; only the error status is passed.

## Preparing to Receive Data from the Device

Issue a PUTLONG to set up the following parameters in preparation for receiving data from the device:

- Intercharacter timeout value. Choose a value compatible with your device. The default value is 100 milliseconds.
- Enable receive (byte CC, bit 2=1).
- For a 4683 terminal feature expansion card I/O port, condition Data Set Ready (DSR) (byte CC, bit 1) and Clear To Send (CTS) (byte CC, bit 5) according to the requirements of your device.
- For a 4683 terminal feature expansion card I/O port, Data Terminal Ready (DTR) must be active to receive data (byte CC, bit 6). Choose this parameter according to the requirements of your device.
- For an I/O port not on a feature expansion card, condition DTR (byte CC, bit 1) and Request To Send (RTS) (byte CC, bit 5) according to the requirements of your device.
- For an I/O port not on a feature expansion card, DSR must be active to receive data (byte CC, bit 6). Choose this parameter according to the requirements of your device.

## Waiting for Received Data

Your application should issue a WAIT statement to wait for data from the device. In most cases you need to wait for data from multiple sources such as the serial device and the I/O processor. The WAIT statement allows you to wait for input from multiple devices including a timer to indicate no input received during a specific time. When good data is available from the device, the statement following the WAIT is performed. When you wait on multiple devices, use the EVENT% statement to determine if the serial device was the device satisfying the WAIT. If an error occurs during a WAIT, control is given to your ON ERROR routine.

Any of the following conditions causes a WAIT to be satisfied by the serial I/O driver:

- The intercharacter timer expires.
- The 247-byte system buffer is full.
- One of the following errors or events occurs:
  - A framing error is detected. A framing error results when the asynchronous character stop bit is not detected at the proper time during reception of a character.
  - A parity error is detected. Feature Expansion hardware error is detected.
  - A break is received from the device.
  - For a 4683 feature expansion card I/O port, DTR is inactive when DTR monitoring is requested with a PUTLONG, and not configured as a current TCC Network.
  - For an I/O port not on a feature expansion card, DSR is inactive when DSR monitoring is requested with a PUTLONG.

## Receiving Data from the Device

Issue READ LINE to receive data from the device. READ LINE is a synchronous operation. If good data is available from the device, the READ LINE is completed and the data is placed in your string variable specified on the READ LINE statement. You can use the LEN function to check the number of bytes received. If an error occurred, control is given to the ON ERROR routine.

## Determining Serial I/O Port Status

The GETLONG statement returns the current settings of the control parameters that can be set with PUTLONG. GETLONG also returns status information related to the device and the serial I/O port. The status information contains the following information for a 4683 feature card serial port:

- Data available
- Data lost
- DTR status (at the time GETLONG is performed)
- RTS status (at the time GETLONG is performed)
- Parity error
- Framing error
- Break received
- Transmit buffer empty

The status information contains the following information for 4693 ports (both native and on Dual Async adapters), 4694, and 4683-421 terminal upgrade native serial ports.

- Data available
- Data lost
- DSR status (at the time GETLONG is performed)
- CTS status (at the time GETLONG is performed)
- Transmit buffer empty

## Preparing to Transmit Data to the Device

For a 4683 feature expansion card I/O port, issue a PUTLONG to condition DSR, CTS, and to set the option that causes the driver to wait or not wait for DTR on transmit (byte CC, bit zero). These settings must be according to the requirements of your device.

For an I/O port not on a feature expansion card, issue a PUTLONG to condition DTR, RTS, and to set the option that causes the driver to wait or not wait for DSR on transmit (byte CC, bit zero). These settings must be according to the requirements of your device.

## Transmitting Data to the Device

Issue a WRITE statement to transmit data from your application to the device. A WRITE is an asynchronous operation. The serial I/O driver has one transmit buffer. If all previous write operations are complete when you issue a WRITE statement, the driver fills its transmit buffer and returns control to the statement following the WRITE command. If a previous write operation is not complete when you issue a WRITE, execution of your application is suspended until the previous write operation ends. You can see if a write operation is complete by checking the transmit buffer empty bit in the status returned on a GETLONG statement.

A write operation is complete when any of the following conditions occur:

- All data specified on a WRITE has been sent to the device.
- For a 4683 feature expansion card I/O port, DTR goes inactive for at least 30 seconds and wait for DTR active was set with a PUTLONG statement (byte CC, bit zero=1), and not configured as a current TCC Network.
- A serial I/O feature expansion hardware error has been detected.
- For a 4683 feature expansion card I/O port, RTS goes inactive for at least 30 seconds and not configured as a current TCC Network.
- For an I/O port not on a feature expansion card, DSR goes inactive for at least 30 seconds and wait for DSR active was set with a PUTLONG statement (byte CC, bit zero=1).
- For an I/O port not on a feature expansion card, CTS goes inactive for at least 30 seconds.

If an error is detected on a write operation, control is given to your ON ASYNC ERROR routine.

## Sending a Break to the Device

You can send a break to the device that is attached to an I/O port by specifying *send break* on a PUTLONG statement (byte CC, bit 3=1). You do not have to reset the bit. Each time you issue a PUTLONG statement with the send break bit set, the driver sends a break to the device. The serial I/O feature expansion causes a break by generating 10 consecutive framing errors.

## Simultaneous Receive and Transmit

Your application can perform receive and transmit operations simultaneously by issuing a WRITE, which is performed asynchronously, and then a WAIT to wait for received data.

## Example

This sample program uses the serial I/O interface to print the following 59-byte character set in a rippled pattern on a serial printer or video display using the RS-232-EIA port.

```
"ABCDEFGHGIJKLMNOPQRSTUVWXYZ0123456789&*.;'""@?.,$=!><()-%+##/ "
```

Make the following assumptions:

1. Port 2 = RS-232 EIA INTERFACE
2. 9600 BPS Baud Rate
3. ASYNC DATA FORMAT
  - NO PARITY

- 8 BIT CODE WITH 1 STOP BIT.

```
%ENVIRON T                                !! INDICATE TERMINAL
INTEGER*4  hx%,sx%                         !! DECLARE VARIABLES
INTEGER*2  ANDSP%,EIAP%
ANDSP% = 1                                !! INIT I/O SESSIONS
EIAP% = 12

SUB ASYNC.ERR(RFLAG,OVER)
INTEGER*2 RFLAG
STRING OVER
RFLAG = 0
OVER = ""
hx% = errn
errfx$ = ""
for s% = 28 to 0 step -4
    sx% = shift(hx%,s%)
    the.sum% = sx% and 000fh
    IF THE.SUM% > 9 THEN \
        THE.SUM%=THE.SUM%+55 \
    ELSE \
        THE.SUM%=THE.SUM%+48
    A$=CHR$(THE.SUM%)
    errfx$ = errfx$ + A$
next s%
clears ANDSP%
write #ANDSP%;"err=",err,"    errl=",errl
locate #ANDSP%;2,1
write #ANDSP%;"errf=",errf%,"    errn=",errfx$
wait ;15000
resume
END SUB

ON ERROR GOTO ERRORA                        !! SET "ON ERROR"
ON ASYNC ERROR CALL ASYNC.ERR              !! SET "ON ASYNC ERROR"
OPEN "ANDISPLAY:" AS ANDSP%                !! OPEN AN DISPLAY
CLEARs ANDSP%
WRITE #ANDSP% ; "SERIAL I/O SAMPLE "
WAIT ; 5000                                !! WAIT 5 SECONDS
! -----
! ENABLE PORT 2 (RS232 EIA INTERFACE) AS I/O SESSION 12
! -----
OPEN SERIAL 2,9600,"N",8,1 AS EIAP% BUFFSIZE 80
LOCATE #ANDSP% ; 2,1                        !! LOCATE TO SEC LINE
WRITE #ANDSP% ; "SER I/O PORT 2 OPEN"
! -----
! CONSTANT FOR 59 CHAR. RIPPLE PATTERN CHARACTER SET
! -----
BUF1$="ABCDEFGHJKLMNOPQRSTUVWXYZ0123456789&*.;'""@?.,$=!><()-%+##/ "
BUF2$ = "ABCDEFGHIJKLMNOPQRST"
B1$ = BUF1$ + BUF1$ + BUF2$   !!! 59 + 59 + 20 FOR RIPPLE 80 CHARS
I% = 1                        !!! VARIABLE FOR RIPPLE COUNTER
WRITE #ANDSP% ; "START RIPPLE PRINT"

EXERCISE:
FOR I% = 1 TO 20
    C1$ = MID$(B1$,I%,80)        !!! MSG DATA-RIPLE PATRN
    WRITE # EIAP%; C1$           !!! SEND 1 MSG EACH PASS
NEXT I%
CLOSE EIAP%                     !! CLOSE SERIAL I/O
WRITE #ANDSP% ; "END OF SAMPLE"
CLOSE ANDSP%                    !! CLOSE AN DISPLAY
STOP                             !! STOP PROGRAM
ERRORA:
!!ERROR ASSEMBLY ROUTINE!!
hx% = errn
errfx$ = ""
```



```

for s% = 28 to 0 step -4
    sx% = shift(hx%,s%)
    the.sum% = sx% and 000fh
    IF THE.SUM% > 9 THEN \
        THE.SUM%=THE.SUM%+55 \
    ELSE \
        THE.SUM%=THE.SUM%+48
    A$=CHR$(THE.SUM%)
    errfx$ = errfx$ + A$
next s%
clears ANDSP%
write #ANDSP%;"err=",err," errl=",errl
locate #ANDSP%;2,1
write #ANDSP%;"errf=",errf," errn=",errfx$
wait ;15000
resume
end

```

---

## Tone Driver

This section describes the tone driver and provides guidelines for using it.

### Characteristics

The tone driver has the following characteristics:

- The tone is an output device that provides audible feedback at the terminals.
- Your application controls the occurrence, frequency, and duration of tones.
- The operator can set the volume of the tones to be high or low by keying input to the terminal system functions. The *4690 OS: User's Guide* explains how to set the tone volume.

### Functions Your Application Performs

Your application program can perform the following functions with the tone:

- Specify the tone frequency and duration.
- Turn the tone on or off on request.

### Accessing the Tone

Use the OPEN statement to gain access to the tone device driver.

Use the CLOSE statement to end your application's use of the tone device driver.

### Generating a Tone

Use the WRITE FORM statement to generate a tone. You must specify the frequency of the tone as either low, medium, or high. You must also choose to either generate a tone of a specified duration or control the tone through individual WRITE FORM statements. If you specify a tone duration, the tone driver turns the tone on, waits the specified duration, and turns the tone off. If you do not specify the duration, you must turn the tone on or off with separate WRITE FORM statements. You can set the volume of the tone by specifying HI or LOW as the duration on a WRITE FORM statement.

When you issue a WRITE FORM statement, the request is queued to the tone driver. Control proceeds to the statement following the WRITE FORM unless an error occurs. The driver allows a maximum of one tone request being sounded and two queued requests; it discards any others. If you request the tone ON (duration not specified), your next tone request must be a tone OFF request. Any tone requests other than tone OFF are discarded.

The terminal tone volume can be controlled by using the WRITE FORM statement with HI or LOW as the duration. To set the volume to high, specify HI as the duration. Specify LOW for low volume. In either case, you must also include a valid frequency. A WRITE FORM with a HI or LOW does not produce any



tone, but sets the volume for future tone requests. The tone volume remains in effect until the next HI or LOW is specified or until system function 6 is entered from the keyboard to toggle the tone volume.

When the driver processes a tone request, it ensures that at least 0.2 seconds elapse between completion of the request and processing another one. This pause ensures that queued tone requests do not sound like a single tone.

The I/O processor can also generate tones. You can specify in the input state table that a tone should result from certain input sequences. The I/O processor issues the tone when these input sequences occur. See the *4680 BASIC: Language Reference* for more information on generating a tone.

## Example

This example writes a message to the display, runs a test of the tone device, and then writes another message to the display.

```
%ENVIRON T
! Declare a two-byte integer.
INTEGER*2  DISPLAY
INTEGER*2  TONE
DISPLAY = 1
TONE = 2
ON ERROR GOTO ERR.HNDLR
! Open the display.
OPEN "ANDISPLAY:" AS DISPLAY
CLEARS DISPLAY
WRITE #DISPLAY; "START TONE TEST"
WAIT ; 2000
OPEN "TONE:" AS TONE
CLEARS DISPLAY
WRITE #DISPLAY; "TONE"
WAIT ;2000
WRITE FORM "C4,C3"; #TONE; "0750", "020"
WAIT; 250
WRITE FORM "C4,C3"; #TONE; "1000", "020"
WAIT; 250
WRITE FORM "C4,C3"; #TONE; "1800", "020"
WAIT; 250
CLOSE TONE
CLEARS DISPLAY
WRITE #DISPLAY; "END TONE SAMPLE"
CLOSE DISPLAY
STOP

ERR.HNDLR:
! Prevent recursion.
ON ERROR GOTO END.PROG
! Determine error type
! and perform appropriate
! recovery and resume steps.
END.PROG:
STOP
END
```

---

## Totals Retention Driver

This section describes the totals retention driver and provides guidelines for using it.

## Characteristics

The totals retention driver has the following characteristics:

- 1024 bytes are available for your application to store data that needs to be saved if the 4683 terminal loses power for a prolonged period. 16 KB are available for your application with a 4693/4694.

- Your application can use the totals retention storage area like it does for either a direct file or a sequential file. The following functions are available:
  - Direct Mode
    - Write data to a specified address
    - Read data to a specified address
  - Sequential Mode
    - Write data
    - Read data
    - Specify offset for next read or write
    - Get offset of last record read or written
- For direct or sequential mode access, four bytes consisting of length and cyclic redundancy check (CRC) information are added to each record written. You must include this overhead when calculating record addresses or space requirements. The overhead bytes are not passed to your application when you read a record.

For sequential mode, there is an additional overhead of four bytes for an end-of-file (EOF) marker. This marker is appended to the end of a record on a write. If the offset is not changed, the previous EOF marker is overlaid by each sequential record write.

The maximum record size that your application can write is 60 bytes in direct mode and 56 bytes in sequential mode for a 4683 terminal, or 1020 bytes in direct mode and 1016 bytes in sequential mode for a 4693/4694 terminal. The totals retention driver automatically adds the required overhead bytes to your records.

The application address space for totals retention is 1 through 1024 for a 4683 terminal, or 1 to 16384 for a 4693-xxx and 4694 terminal. Your application has the responsibility to determine the record start and length in direct mode.

## Accessing the Totals Retention Driver

Use the OPEN statement to gain access to the totals retention driver.

The CLOSE statement ends communication with the totals retention driver.

## Accessing Totals Retention in Direct Mode

Direct mode is specified when you use the READ FORM or WRITE FORM statement. You must always specify the address (1 through 1020 for 4683, 1 through 16380 for 4693/4694) of the record to be read or written. You cannot specify direct mode access to start at addresses greater than 1020 for the 4683 terminals or 16380 for the 4693 and 4694 terminals because of the overhead requirements. There is no EOF marker added on a direct mode write.

## Reading Data in Direct Mode

To read a record in direct mode, issue a READ FORM statement specifying the totals retention address of the start of the record. The data (without the overhead bytes) is placed in the variables specified on your READ FORM statement. If there is an error in the data read, control passes to your ON ERROR routine. If the error returned is offline, there is no data put into the specified variable.

## Writing Data in Direct Mode

To write a record in direct mode, issue a WRITE FORM statement specifying the starting address (1 through 1020 for 4683, 1 through 16380 for 4693/4694) of the record. The number of bytes in your record must be 1 through 60 (4683 terminals), or 1 through 1020 (4693/4694 terminals). If an error is detected on the WRITE FORM, control passes to your ON ERROR routine.

## Accessing Totals Retention in Sequential Mode

Sequential mode is specified when you use the READ LINE, WRITE, or POINT statements or the PTRRTN function. For sequential mode, the totals retention driver maintains a pointer to the next record to

read or be written and a pointer to the beginning of the last record that was read or written. After an OPEN statement is performed, both pointers point to address one.

## Specifying the Address in Sequential Mode

When you read or write records in sequential mode, the totals retention driver uses the next record pointer to determine the starting address of the read or write. Following a successful read or write, the next record pointer and last record pointer are updated. Issue a POINT statement when you want to prepare to read or write a record other than the next sequential record. The address on the POINT statement must be between 1 and 1020 (4683 terminals) or 1 and 16380 (4693/4694 terminals) for a READ LINE. The address on the POINT statement must be between 1 and 1016 (4683 terminals) or 1 and 16376 (4693/4694 terminals) for a WRITE.

If you want to access the last record read or written, use the PTRRTN function to return the address of the last record accessed. This address can be used on a POINT statement to prepare to reread or rewrite the last record accessed.

## Reading Data in Sequential Mode

Issue a READ LINE statement to read a record in sequential mode. The next record pointer contains the starting address for the read. Following the read, the record data is placed in the string variable specified on the READ LINE statement. If an EOF marker is detected or an error occurs, control passes to your ON ERROR routine.

## Writing Data in Sequential Mode

Issue a WRITE statement to write a record in sequential mode. The next record pointer contains the starting address for the write. Following a successful write, the next record pointer and last record pointer are updated and an EOF marker is written.

When you write records sequentially by letting the next record pointer indicate the address, the previous EOF marker is overwritten and replaced by the new record and new EOF marker. If you use the POINT statement to change the address of the next write, it is possible to create multiple EOF markers in Totals Retention storage.

## Example

The following example contains code written to communicate with a totals retention driver. This example writes a message to the display, runs a test of the totals retention driver by writing data to it and then reading data from it. It then compares the data that it stored with the original data, writing messages about the results of the comparison to the display.

```
%ENVIRON T
! Totals Retention sample.
! Define the variables.
INTEGER*2 HARDTOT, DISPLAY,I1,I2
INTEGER*2 J1,J2,OFFSET
INTEGER*4 I3,I4,J3,J4,TIME
REAL R1,R2
STRING DATA1$,DATA2$,RDMFMT$
! Set up the error handler.
ON ERROR GOTO ERROR1
! Open the totals retention driver and
! the AN display for messages.
HARDTOT = 1
DISPLAY = 2
TIME = 5000
OPEN "ANDISPLAY:" AS DISPLAY
OPEN "TOTRET:" AS HARDTOT BUFFSIZE 1020
CLEARS DISPLAY
WRITE #DISPLAY; "START OF TOTALS RETENTION SAMPLE"
WAIT ; TIME
```

```

! Initialize the variables.
OFFSET = 1
R1 = 100.3
I1 = 1
I2 = 2
DATA1$ = "ABCDEFGHIJKLMNOPQRSTUVWXYZ7890"
I3 = 35
I4 = 36
RDMFMT$ = "R 2I2 C30 2I4"
! Write the data in the totals retention area of storage.
WRITE FORM RDMFMT$;#HARDTOT,OFFSET;R1,I1,I2,DATA1$,I3,I4
CLEARS DISPLAY
WRITE #DISPLAY; "WRITE COMPLETE
WAIT ;TIME

! Then read the data back, and compare with the original
! data. This portion of the example could be performed
! after a power-down situation to demonstrate totals
! retention.
RDMREAD:
READ FORM RDMFMT$;#HARDTOT,OFFSET;R2,J1,J2,DATA2$,J3,J4
! Compare the data read with original data.
CLEARS DISPLAY
IF R1 = R2 THEN \
    WRITE # DISPLAY; "REAL DATA COMPARES OK"\
ELSE \
    WRITE # DISPLAY; "REAL DATA DOESN'T COMPARE OK "
WAIT ; TIME
CLEARS DISPLAY
IF I1 = J1 THEN \
    WRITE # DISPLAY;"INTEGER DATA COMPARES OK -1 " \
ELSE \
    WRITE # DISPLAY; "INTEGER DATA DOESN'T COMPARE OK -1 "\
WAIT ; TIME
CLEARS DISPLAY
IF I2 = J2 THEN \
    WRITE # DISPLAY; "INTEGER DATA COMPARES OK -2 " \
ELSE \
    WRITE # DISPLAY ;"INTEGER DATA DOESN'T COMPARE OK -2"
WAIT ; TIME
CLEARS DISPLAY
IF I3 = J3 THEN \
    WRITE # DISPLAY;"INTEGER DATA COMPARES OK -3 " \
ELSE\
    WRITE # DISPLAY;"INTEGER DATA DOESN'T COMPARE OK -3"
WAIT ; TIME
CLEARS DISPLAY
IF I4 = J4 THEN \
    WRITE # DISPLAY;"INTEGER DATA COMPARES OK -4 " \
ELSE \
    WRITE # DISPLAY;"INTEGER DATA DOESN'T COMPARE OK -4"
WAIT ; TIME
CLEARS DISPLAY
IF DATA1$ = DATA2$ THEN \
    WRITE # DISPLAY;"CHARACTER DATA COMPARES OK " \
ELSE \
    WRITE # DISPLAY:"CHARACTER DATA DOESN'T COMPARE OK "
WAIT ; TIME
CLEANUP:
CLOSE HARDTOT
CLEARS DISPLAY
WRITE # DISPLAY;"END OF TOTALS RETENTION SAMPLE "
CLOSE DISPLAY
STOP

ERROR1:
! Prevent recursion.

```

```
ON ERROR GOTO ERROR2
! Determine error type and perform appropriate recovery
! and resume steps.
ERROR2:
STOP
END
```



---

## Chapter 4. Using error recovery procedures and facilities

This chapter shows you how to use error recovery procedures and facilities.

---

### Error recovery options

The operating system provides procedures and facilities that help you recover from error situations or from situations that could result in the loss of data. These procedures are either:

- Automatic functions of the operating system
- Enabled during configuration
- Included in your applications

The operating system provides several means for recovering from errors:

- 4680 BASIC functions and statements
- A facility for logging system errors or events
- Special write operations that help save your data during a power line disturbance
- A storage retention function on your terminal that protects data during a power outage

---

### Error functions and statements

Several 4680 BASIC functions and statements help you process and recover from errors associated with your application. These include the ON ERROR, ON ASYNC ERROR CALL, IF END#, and RESUME statement, and the ERR, ERRL, ERRF%, and ERRN functions. See the *4680 BASIC: Language Reference* for additional information on each of these statements and functions.

### ON ERROR statement

The ON ERROR statement specifies a label where control is given when an error associated with a synchronous operation occurs. This statement should be the first executable (nondeclarative) statement in your main application. All operations that your program generates are synchronous operations except for writing data to the terminal printer, using the serial I/O driver, and writing data to a host communication session or link.

Your ON ERROR routine can contain any valid 4680 BASIC statement or function. The first statement in your ON ERROR routine should be a test to see if you have run out of memory (ERR=OM). If you get this error, do not attempt to initialize or increase the size of a string or an array. This would result in another OM error and your program would end up in an infinite loop.

Your application can have multiple ON ERROR statements. However, one ON ERROR routine is in effect at any one time. If your application consists of multiple functions and subprograms, each of these can contain its own ON ERROR routine. When you enter the function or subprogram and an ON ERROR statement is executed, the runtime library remembers the previous ON ERROR routine that was in effect. When you exit the function or subprogram, the runtime library restores the ON ERROR routine to the label it had before entering the procedure.

The ON ERROR routines in the terminal should check to determine if the terminal experiencing the I/O error is powered off. The applications for the Mod1 and Mod2 terminals both execute when the Mod1 terminal is powered on. Because Mod2 might be powered off while the Mod1 terminal application is still running, the ON ERROR routine should always check whether the I/O error was caused by the powered-off terminal. You should use the ADXSERVE function for the application status to determine whether the terminal is powered on or powered off. ADXSERVE always indicates that the Mod1 terminal is powered on. See “ADXSERVE (requesting an application service)” on page 293 for more information on the ADXSERVE function.

When you have determined the type and source of the error, you must issue a RESUME statement to recover from the error.

When a runtime error occurs during an I/O operation, that I/O session number should not be closed in the ON ERROR code. Closing it before resuming from the error leaves runtime data pertaining to that session number in an indeterminate state. If a file, pipe, device or communication link must be closed because of an error, the ON ERROR code should set a flag to be checked in the mainline code where the I/O session can be safely closed.

## **ON ASYNC ERROR CALL statement**

The ON ASYNC ERROR CALL statement specifies a subprogram where control is given when an error associated with an asynchronous operation occurs. This statement should be the second executable (nondeclarative) statement in your main application if it contains any asynchronous operations. Asynchronous operations include writing data to the terminal printer and serial devices and writing data to a host communication session or link.

Your ON ASYNC ERROR CALL subprogram can contain most valid 4680 BASIC statements and functions. This subprogram must have its own ON ERROR routine, local to this subprogram, to handle any synchronous errors that occur when executing this recovery path.

Your application can have multiple ON ASYNC ERROR CALL statements. Only one ON ASYNC ERROR CALL subprogram is in effect at any one time, however. If your application consists of multiple functions and subprograms, each of these can contain its own ON ASYNC ERROR CALL subprogram. When you enter the function or subprogram and an ON ASYNC ERROR CALL statement is executed, the runtime library remembers the previous ON ASYNC ERROR CALL subprogram that was in effect. When you exit the function or subprogram, the runtime library restores the ON ASYNC ERROR CALL subprogram to the label it had prior to entering the procedure.

The ON ASYNC ERROR CALL subprograms in the terminal should check to determine whether the terminal experiencing the I/O error is powered off. See “ON ERROR statement” on page 153 for information on using the power on or off indication.

See the *4680 BASIC: Language Reference* for information regarding the definition of the subprogram and returning control to the application where the error occurred.

## **IF END# statement**

The IF END# statement specifies a label where control is given when a file-not-found, end-of-file, or disk-full error occurs for a specified sequential, random, or direct file. For a keyed file, the label gets control when a file-not-found or record-not-found error occurs.

An IF END# statement, unlike the ON ERROR and ON ASYNC ERROR CALL statements, is associated with only one file. You can have only one IF END# statement active for each file; but you can have several files active, each with its own IF END# statement.

Your IF END# routine can contain any valid 4680 BASIC statement or function.

When you have taken some corrective action in your IF END# routine, you must issue a GOTO statement to return to some point in your application.

## **RESUME statement**

The RESUME statement enables some type of recovery from all synchronous errors. The recovery options available depend on the type of error that occurred. There are two types of errors: I/O errors and non-I/O



errors. The options available for recovery from an I/O error are RESUME, RESUME RETRY, and RESUME label. The only option available for non-I/O errors, which are usually logic errors, is RESUME label.

**Note:** The RESUME statement may only be executed within an ON ERROR routine. Do not place the RESUME statement in a subprogram, function, or subroutine that is called by the ON ERROR routine.

## RESUME

Execution of a RESUME statement causes the failing I/O statement to be bypassed. This ignores a noncritical I/O instruction that caused an error and continues execution at the next sequential instruction.

## RESUME RETRY

Execution of a RESUME RETRY statement causes the failing I/O statement to be executed again. The application must keep track of the number of times the statement is executed to avoid an infinite loop if there is a hard I/O failure.

## RESUME label

Execution of a RESUME label statement causes the failing statement to be bypassed and control to be given to a specified point in the application. You might want to designate several key recovery points in your application where control should be given in case of a logic error or some other unrecoverable error. You are able to branch to one of several recovery points in your application, depending on how far you have gotten through your program when the error occurs. You can also call an application service to take a system dump so that debugging data is available. Do not RESUME to a label that identifies a subroutine.

## ERR function

The ERR function returns a two-character string containing the error message of the last runtime error that occurred. If no error has occurred when ERR function executes, it returns a null string. To identify and get an explanation of the two-byte error code, see the *4680 BASIC: Language Reference*.

## ERRL function

Use the ERRL function when debugging your application. It returns the line number at which the error occurred. If no error has occurred when this function executes, it returns a zero.

## ERRF% function

The ERRF% function returns the I/O session number associated with the most recent I/O error.

## ERRN function

The ERRN function returns a four-byte error code associated with I/O and operating system errors. This error code can help isolate the specific cause of an error when multiple conditions exist that could generate a single error code.

---

## Logging system errors

The operating system handles error recovery through retry operations and event or error logging. If the operating system detects an error, it usually retries the operation automatically. It continues system operation unless data integrity is endangered. Events and errors that take place on the system are recorded in a system log.

## System log

The operating system provides an event and error logging function that gives you a record of events surrounding a system error. You can use the data collected by this function to help resolve problems that you identify after a logged event or error takes place.

The data collected by the event or error logging function is stored in a set of files called the *system log*.

This system log consists of six files in which the new entries replace the old entries. The six files in the system log are:

- Store Controller Hardware Errors
- Terminal Hardware Errors
- Terminal Events
- Store Controller Events
- System Events
- Application Events

Placing system log data into these files helps isolate serious log entries (for example, a broken terminal printer) from entries resulting from expected occurrences (for example, a terminal recovering from a power line disturbance). The system uses the following guidelines for selecting a file for a particular event or error:

File	Use
Store Controller Hardware Errors	Used for faults that can be corrected by hardware repair at the store controller.
Terminal Hardware Errors	Used for faults that can be corrected by hardware repair at the terminal.
Terminal Events	Used for recording errors or events in terminals.
Store Controller Events	Used for recording errors or events in the controller.
System Events	Used for a wide variety of normal events (for example, initial program loads (IPLs), PLD recoveries), program-induced faults (for example, program checks), logical environment faults (for example, missing data sets), or certain operator-induced events (for example, choosing a system application).
Application Events	Used for events generated by application programs. The application can execute in either the terminals or store controller.

## System messages

Your operating system uses *system messages* to communicate exception conditions. These messages result from internal conditions that the operating system detects and logs as events or errors in the system log. In addition, system messages can result from your application calls to ADXERROR to accomplish appropriate application event logging.

System messages give information about errors and tell you the action needed to correct such errors. The messages consist of an identifier and descriptive text.

### System messages at the store controller

The main operator interface program handles system message requests in the store controller. This program receives requests, formats and writes system log entries, manages the six system log files, and based on severity code, adds messages to the SYSTEM MESSAGE DISPLAY panel.

The general format for messages displayed at the store controller is:

```
mm/dd hh:mm cc ttt s annn xxxxxxxxxxxxxxxxxxxxxxxx
                        xxxxxxxxxxxxxxxxxxxxxxxx
```

where:

*mm/dd hh:mm*

is the log entry time stamp.

*cc* is the controller identifier.

*ttt* is the terminal number.

`s` is the severity code for the event or error.  
`annn xxx...xxxxx`  
is the message text.

For more specific information about the content and use of system messages, see the *4690 OS: Messages Guide*.

## Severity codes

You can specify which messages you want displayed on the SYSTEM MESSAGE DISPLAY panel through the use of *severity codes*. Depending on the severity code you set, messages at or below a certain severity level are shown as they are logged.

Severity codes are assigned to system message entries. Use these codes to indicate the impact of certain events or errors on a store's operation. For information on the severity codes, see the *4690 OS: Messages Guide*.

For information on setting the system message display level, see the *4690 OS: User's Guide*.

## Severity codes at the terminal

System messages displayed at the terminal consist of a system message identifier and text and pertain only to problems the terminal operator needs to know about for recovery. These messages originate only from conditions detected by the operating system.

Terminal Services, an interface program for terminal software, provides logging and system message support for the terminal. Events or errors in each terminal component are reported to Terminal Services by specifying a message number, a group character, a request type, and any error or event parameters. When Terminal Services receives error or event data, it sends the data to the store controller, and the data can be displayed if wanted. If a terminal has more than one display, the Terminal Services messages is displayed on the system display. Depending on the request type specified, the message is displayed immediately or delayed. Delayed messages can be viewed only by entering key sequences that display the message. For information on system message display operations at the terminal, see the *4690 OS: User's Guide*.

The format of the terminal message depends on the type of display attached to the terminal. For information on terminal message format, see the *4690 OS: Messages Guide*.

---

## Event log access

The operating system provides a function to give your program realtime access to event log information. This is accomplished by having the operating system create a pipe and write event log data into this pipe. Your program can then open the pipe and read the data from the pipe.

To get realtime access to event log information, use controller configuration to define a user logical name (ADXCSOUS) with a value for the pipe size. The minimum size for the pipe is 128 bytes; the maximum size is 65,536 bytes. If a value outside this range is specified, the value is rounded up to 128 or down to 65,536. The size specified is also rounded up to the next higher multiple of 32 if necessary.

Based upon this define, the operating system creates a pipe (ADXCSOUP) and writes from the error logging buffer into this pipe. Your application should open this pipe and read 32-byte records out of the pipe in order to obtain event logging data.

If for any reason the operating system receives an error writing event log data to the pipe, a dummy record containing the number of records missed because of the write failure is written into the pipe.

**Note:** If this function is used on the master store controller or alternate master store controller, events logged by all store controllers on the LAN are written to the pipe by the operating system. If this function is used on a subordinate store controller, only events logged by the subordinate store controller are written to the pipe.

The operating system writes the events to the pipe as soon as the pipe is created, which is early in the IPL process of the controller. However, your application is not started until the end of the IPL process, so many events can be written to the pipe before the application starts to read the pipe. This should be taken into consideration when selecting the size for the pipe.

Also, if a store controller has been disconnected from the LAN for some period of time, it rapidly sends event log data to the master store controller when it is reconnected. This action should also be taken into account when selecting a size for the pipe. Your application can keep track of the number of records not written to the pipe by using the counts contained in the dummy records written by the operating system. The pipe can be enlarged, if necessary, until dummy records are no longer encountered.

## Record formats

The information in the following table is the format of the dummy record written to the pipe after errors have occurred while writing event log records to the pipe. A dummy record most likely occurs if the pipe is allowed to fill up and the operating system can no longer write records to the pipe until your application reads the pipe and makes room for more data.

Offset	Length	Type	Content
0	2	UWORD	Reserved
2	2	UWORD	Number of missing records
4	28	UBYTE	Reserved

where UBYTE is an unsigned char and UWORD is an unsigned short. The *reserved* fields will contain zeroes.

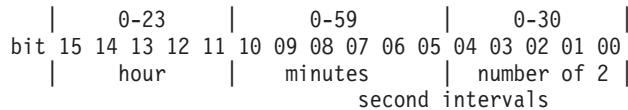
Following is the format of the record that is written to the pipe each time an event is logged:

Offset	Length	Type	Content
0	2	UWORD	Date (see below for format)
2	2	UWORD	Time (see below for format)
4	2	UWORD	Controller Node ID
6	2	UWORD	Terminal Address
8	1	UBYTE	Source
9	1	char	Message group (for example, W507 — message group = W)
10	2	UWORD	Message number (for example, W507 — message number = 507)
12	1	UBYTE	Severity
13	1	UBYTE	Event
14	18	UBYTE	18 bytes of unique data (not formatted)

## Date format

	0-119 (1980-2099)		1-12		1-31	
bit	15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00					
	year		month		day of month	

## Time format



---

## Audible alarm

The operating system provides a function to sound an audible alarm at the store controller when system messages are logged. This alarm alerts operators to situations that might require immediate action to keep store operations running smoothly.

You can activate the alarm for a specified length of time, deactivate the alarm, and report the status of the alarm. The functions can be used locally or remotely from a host processor.

You can select a report on the SYSTEM MESSAGE AUDIBLE ALARM FUNCTIONS panel to indicate the status of the audible alarm at each controller. The report includes the store controller ID (if it is activated), the length of time the alarm is set to sound, and informs you if the alarm is activated. The message numbers can be displayed, printed, or written to a file.

## Setting the audible alarm function

The audible alarm function consists of four parts:

- Building the list of messages to sound the alarm
- Reporting the list of messages to sound the alarm
- Activating the function
- Deactivating the function

To set the audible Alarm, select the Installation and Update Aids option from the SYSTEM MAIN MENU. Then select the System Message Audible Alarm option on the INSTALLATION AND UPDATE AIDS panel.

To build a list of messages to sound the alarm, type **1** from the SYSTEM MESSAGE AUDIBLE ALARM FUNCTIONS panel. A table appears where you can enter message numbers. After determining which messages you want to activate the audible alarm, enter the message numbers in the table on the panel. If you have more message numbers than can fit on one screen, press **PgDn**. Another panel appears that allows you to enter numbers. You can enter up to 100 message numbers. After you have entered the appropriate message numbers, press **Enter** to save the numbers.

**Note:** The message numbers you select apply for all store controllers on a LAN (MCF Network).

To report the list of messages selected, select **2** from the SYSTEM MESSAGE AUDIBLE ALARM FUNCTIONS panel. You can direct the report output to the screen, the printer, or to a file by placing a value of 1, 2, or 3 in the Destination of Output field on the panel. If you select to direct the report output to a file, the report is placed in ADX\_SDT1:ADXCS1RF.DAT.

Message numbers are four character spaces in length. The first space is a character (A, B, C, D, T, U, W, X, Y, or Z). The next three spaces are numbers from 000 to 999. See the *4690 OS: Messages Guide* for an explanation of these messages.

## Setting the audible alarm function through RCP

The audible alarm function can also be set using the Remote Command Processor (RCP). See the *4690 OS: Communications Programming Reference* for more information on the Remote Command Processor.

You cannot perform the Build Audible Alarm Message File function through RCP. You must build the audible alarm message file at your central site store controller. The file containing the messages that sound the alarm is ADX\_SDT1:ADXCS1AF.DAT. After building the file at your central site store controller,

you must send this file to your central site host processor. You can then transmit the file to each remote store controller. The RCP commands enable you to perform the report, activate, and deactivate functions directly.

For information on command formats using RCP, see the *4690 OS: Communications Programming Reference*.

## Activating the audible alarm function

Option 3 on the SYSTEM MESSAGE AUDIBLE ALARM FUNCTIONS panel allows you to activate the audible alarm function. Unless you activate the function, the alarm does not sound for the chosen messages. If you want the alarm to sound on more than one store controller in your 4690 LAN (MCF Network) but not on all of the controllers, you can use the activate function as many times as you have store controllers to activate by changing the node ID.

You can limit the length of time the alarm sounds when a chosen message is logged. In the Alarm time limit in seconds field, you can enter an asterisk (\*) indicating that the alarm sounds continuously until the operator views the messages on the SYSTEM MESSAGE DISPLAY panel. You can also enter a numeric value from 0 to 99 in this field. If a numeric value is entered, the alarm sounds for that number of seconds.

The message is highlighted on the SYSTEM MESSAGE DISPLAY panel even if the alarm has finished sounding.

In the Alarm Based on System Message Level field, you can indicate which of the two ways a message alarm can be sounded. This option is important because a message can be logged with different severity levels at different times. When enabling the audible alarm function, you should decide under which of the following two conditions the alarm is to be sounded:

1. The first condition is by message number only. This means the alarm is sounded only if the logged message number matches a message number that you selected.
2. The second condition is by message number and message severity. This means that the alarm is sounded only if:
  - The logged message matches a message number that you selected and,
  - The message has a severity level equal to or higher than the current system message level setting (1 is the highest severity and 5 is the lowest severity).

The severity of a message is an indicator of the severity of the event. Messages with severity 1 indicate that a serious error has occurred and some action is required. Messages with severity 5 indicate something of interest has happened, but requires no action on the part of the operator. Refer to the "Severity codes" on page 157 for more information on setting the System Message Display Level and also see the *4690 OS: User's Guide*.

## Using the audible alarm

When the alarm sounds, the operator must press **Sysreq** and **M** on the store controller keyboard to view the SYSTEM MESSAGE DISPLAY panel. The message causing the alarm is highlighted on the SYSTEM MESSAGE DISPLAY panel. Pressing the **Sysreq** and **M** keys stops the alarm. The duration of the alarm is determined by either of the following:

- The length of time that the user specifies
- The store controller operator views the SYSTEM MESSAGE DISPLAY panel by pressing the **Sysreq** and **M** keys.

If you are operating a 4690 system with the Multiple Controller Feature on a LAN, you can request that the report be generated for all store controllers on the LAN or a specific store controller. You can request this by placing an asterisk (\*) or a two-character store controller ID in the ID of controller field on the panel.

The operator should follow the operating procedures provided by store personnel for each of the messages that are highlighted. You can enable or disable this function at any time.

## Deactivating the audible alarm function

You can deactivate the audible alarm function by selecting option **4** on the SYSTEM MESSAGE AUDIBLE ALARM FUNCTIONS panel. Deactivating the function prevents the alarm from sounding and prevents the messages from being highlighted on the SYSTEM MESSAGE DISPLAY panel.

---

## Distribution exception log

On a LAN (MCF Network), all unsuccessful requests made to other nodes are entered in the Distribution Exception Log. For more information on this log, which exists on both the master and file server nodes, see the *4690 OS: User's Guide*.

---

## Power line disturbance recovery

A PLD is an interruption in the AC power to the store. The duration of the PLD determines the effect on the operation of the store controller and the terminal. Some PLDs are short enough that there is no effect; others cause the unit to stop operation.

When a PLD causes the store controller to shut down, the store controller re-IPLs after power is restored. When a PLD causes the Mod1 or Mod2 terminals to stop, the storage retention function supplies power to the memory of the terminal.

## Store controller PLD recovery

The operating system provides four levels of PLD recovery for the disk functions. The levels are for a sector, the File Allocation Table, a single record, and a record pair. See “Performing file functions” on page 29 for a description of the single record and record pair PLD recovery facilities.

The file PLD recovery facility uses a battery-powered NVRAM to save the data to recover the file functions interrupted by a PLD.

Sector level recovery saves the sector of data to be written to the disk for each disk write. If a PLD prevents a sector from being written, the sector is written as part of the store controller IPL.

File Allocation Table recovery keeps track of whether copy 1 or copy 2 of the File Allocation Table is being updated. The copies are updated serially. If a PLD occurs during the update of either copy of the File Allocation Table, the system copies the unaffected copy to the affected copy as part of the store controller IPL.

You should write your store controller applications so that they can be restarted after a PLD. File updates should be done so that they can always be restarted or the updates should have checkpoints.

## Terminal PLD Recovery

Terminal PLD recovery is dependent on the storage retention function. If the PLD does not last longer than the batteries provide power to the terminal memory, the terminal application restarts where it stopped. Your application needs to have I/O device error recovery procedures, because any I/O function interrupted by the PLD indicate a failure or a timeout error. The other functions of the application should not be affected by the PLD. The operating system restores the data on the displays after a PLD. Because the terminal application is ready to start when the power is restored and the store controller is IPLed, the terminal application may want to allow the terminal operator to wait for the store controller to become available instead of continuing without the store controller. See “Storage retention” on page 162 for details on the storage retention function.



---

## Storage retention

Storage retention enables various power modes depending on the machine type and the function performed. When enabled from the controller system panel, an inactivity time can be specified when a terminal will go into a lower power state (sleep mode) when the time expires. Also while enabled a programmable power off command will put the terminal into a lower power state rather than power off. Finally, the machines that have a backup battery capability will switch to the backup battery state when AC power is lost.

**Note:** Storage retention is not supported on the terminal side of a controller/terminal.

- | The battery provides the power to the memory. The actual length of time depends on the amount of
- | memory, the size of the battery, and the battery charge. If the power outage lasts longer than the battery
- | can power the memory, the original contents of memory and function in process are lost and the terminal
- | re-IPLs when the power is restored. Some SurePOS machines can power I/O during this time. Some
- | SurePOS and TCxWave 6140 Series machines can power I/O during this time.

You can enable and disable the storage retention function from the system panels, a store controller application, a terminal application, or a terminal keyboard.

If the terminal operating system or application is in a code loop, powering off the device and then powering on with storage retention enabled only returns to the looping condition. See the system request functions in the *4690 OS: User's Guide* for information on getting out of the loop.

**Note:** The 4694 Point-of-Sale Terminal must be equipped with the battery backup feature to support storage retention.

---

## Terminal power on/off

This section describes the power on/off functions for the Mod1 and Mod2 terminals.

### Terminal power on/off for the Mod1 terminal with storage retention enabled

If power remains at the wall plug for the Mod1 terminal, powering off the terminal interrupts power to all of the terminal functions except the memory. The power off is treated as a PLD by the operating system. As long as power remains at the wall plug, memory is retained and the storage retention batteries are recharging. If the power is interrupted at the wall plug while the storage retention feature is enabled, the battery powers the memory until power is restored or until the battery discharges.

When the terminal is powered on and the memory is still retained, the application is restarted by the same process used in a PLD recovery.

### Terminal power on/off for the Mod2 terminal

The power-off switch on the Mod2 terminal interrupts power to all the Mod2 terminal functions. It does not affect the memory in the Mod1 terminal that is used for the Mod2 terminal. If the Mod1 memory is retained switching the Mod2 power off and on appears to the Mod2 terminal as a PLD recovery situation. On the 4693 terminals, if the Mod1 is turned off, the Mod2 automatically powers down in about 7 seconds.

---

## Disk and file error recovery

Your application's ON ERROR routine must provide error recovery for the disk functions. Your application can also use the IF END# statement.



Several types of errors can occur when a disk I/O statement is performed. Some of the errors indicate that your program has attempted a function that is not valid; others indicate that there is a problem with the operation of the disk or a problem with a particular file on the disk. Only those errors that are associated with the disk and files are described in this section.

Your error handling routines should be different for the terminal and the store controller. For example, in some situations you might want to design the terminal application to function in a stand-alone mode when access to the store controller is lost.

You can use the IF END# statement to detect end-of-file, missing records in keyed files, missing files, and disk-full situations. See the *4680 BASIC: Language Reference* for the details of using the IF END# statement. Your application can either use a different IF END# routine for reads, writes, and so on, or must set a variable to indicate to the IF END# routine the file function attempted.

All other disk and file errors cause the ON ERROR routine to be given control. If an IF END# statement is not used, the IF END# types of errors also cause the ON ERROR statement to be given control. IF END# statements are defined on a unique file basis and an ON ERROR statement is applicable to all disk and file errors.

When errors detected on a READ statement cause the application to start the ON ERROR routine, all of the variables specified on the failing READ statement are set to zero or null when a RESUME statement is performed.

Table 16 on page 163 shows errors that can occur for some or all of the disk 4680 BASIC statements. Only the most likely errors needing application recovery code are listed. Most of the errors not listed in this section could be treated as your application would treat other program defects. You should review all of the possible disk and file errors by seeing the *4680 BASIC: Language Reference*.

Table 16 on page 163 shows errors that apply to both the store controller and the terminal. The xx values indicate values that you need not pay attention to.

*Table 16. Errors on the store controller and terminal*

Error Code	Description	Action Type
EF 0000001C	No IF END statement and attempt to read past end of sequential file	Type 12
EF 0000001C	No IF END statement and no record found for keyed, random, direct	Type 16
DW 00000018	No IF END statement and there is no disk or directory space to write another record	Type 20
ME 0000003C	No IF END statement and there is no disk or directory space to create a file	Type 20
xx 80xx4001	Cannot open file due to access rights	Type 10
NR 80xx4007	Bad FNUM	Type 24
xx 80xx4010	File not found	Type 14
xx 80xx400C	Cannot access file due to current use	Type 11
xx 80xx400D	Not enough memory	Type 17
xx 80xx4301	Media change occurred	Type 15
OM 80F30000	Insufficient memory	Type 17
KF 80F306C6	Chain in keyed file is not valid	Type 13
EF 80F306C8	Record not found in keyed file	Type 13
DW 80F306CE	Keyed file is full	Type 13

Table 17. Errors unique to the terminal

Error Code	Description	Action Type
XS 0000042C	AUTOUNLOCK of a WRITE statement failed for a random or direct file	Type 22
XT 0000042E	AUTOLOCK of a READ statement failed for a random or direct file	Type 23
OM 80F10000	Insufficient memory	Type 18
TO 80F10681	Terminal offline	Type 21
TF 80F206A1	No files opened	Type 19
TF 80F206A2	No read-only files opened	Type 19
TF 80F206A3	Temporarily cannot process this file request	Type 19
TF 80F206A4	No more files opened	Type 19

The following list describes the type of action for your application. For all of the following, your application should report the error to the operator or log the error for later analysis. Avoid redundant logging.

#### Action Type

##### Explanation

- 10** Your application is requesting access rights that cannot be satisfied. This is probably a file security problem. Your application should report that the user is trying to access a file for a function that is not authorized.
- 11** Your application is requesting access to a file and it is in conflict with the current usage of the file.
- 12** Record specified by application was beyond the current end-of-file.
- 13** The keyed file has a problem that requires user attention.
- 14** The file does not exist as defined by the name.
- 15** The user changed the diskette. Prompt the user to restore the original media and continue.
- 16** The record specified for a READ does not exist.
- 17** There is insufficient dynamically allocatable memory in the store controller to perform the requested function.
- 18** There is insufficient dynamically allocatable memory in the terminal to perform the requested function.
- 19** The applicable Shared I/O Access Method table was full.
- 20** The disk drive is full and requires user attention to free disk space.
- 21** No files are available in the store controller. The application should execute in offline mode if possible. When the store controller is available, the system opens the files again. The application should not assume that previous file functions were successful. For example, the application should repeat a READ AUTOLOCK instead of using the results of a previous READ AUTOLOCK prior to executing a WRITE AUTOUNLOCK.
- 22** This could be a program defect; the program is trying to unlock a record that it does not have locked. It could also occur if the terminal temporarily loses communication with the store controller. The application should repeat the READ with AUTOLOCK and WRITE with AUTOUNLOCK sequence for this record.
- 23** The file record is already locked. Your application should wait and retry. If the condition persists, notify the user that this record cannot be accessed.
- 24** The file is no longer able to be accessed because of a bad FNUM. The file should be closed and

then reopened again by the application. Caution should be taken so that the application only tries to reopen the file a selected number of times so that a suspend condition does not occur due to looping.

---

## Error recovery for I/O devices

Many error conditions can occur when you use terminal I/O devices. Some errors indicate that your program has attempted a function that is not valid; others indicate that there is a problem with the device.

### Application responses

The type of response your application makes to an error condition depends on the type of error that occurred. Depending on the type of error, your application must perform one of the following:

***Request operator intervention.*** Your application asks the operator to take corrective action by putting a message on a device other than the failing one. For example, if the error is caused by a paper jam, the application could write a message to the display requesting that the operator clear the jam. The application could then wait for the operator to signal (usually by a keyboard entry) that corrective action has been taken.

***Retry the operation.*** Your application requests that the failing I/O operation be retried. If the error caused entry to the ON ERROR routine, the retry is done by a RESUME with the RETRY parameter. For a terminal printer error, the retry occurs when you set the retry flag parameter on (set to -1) in the ON ASYNC ERROR subprogram and then exit the subprogram.

***Bypass the operation.*** Your application requests that the failing I/O operation not be retried. If the error caused entry to the ON ERROR routine, the bypass is done by a RESUME without the RETRY parameter or a RESUME with a label specified. If the error caused entry to the ON ASYNC ERROR subprogram, the bypass is done when you exit the subprogram with the retry flag parameter off (set to 0).

See the *4690 OS: Messages Guide* for error codes for I/O devices.



---

## Chapter 5. User Application Considerations with a LAN (MCF Network)

The chapter describes the user application considerations with a LAN (MCF Network).

---

### Using TCLOSE to Close Application Data Files

Because applications open and close application data files, you should be aware of each file's distribution attributes when programming the updating and closing of files.

If the file mode attribute is Distribute At Close, and the application requires that a snap-shot of the file be taken periodically so that the image version closely matches the prime version, you should be very cautious about how often TCLOSE is used. You should be especially cautious if the file is large because TCLOSE can take considerable time distributing a large file over the LAN (MCF Network). See the *4680 BASIC: Language Reference* manual for a discussion of the TCLOSE instruction.

While TCLOSE permits more frequent distribution of the file without requiring both a close and an open, it could also have a significant performance impact on the system because of the resulting network traffic.

**Note:** A checkout application has priority over the distribution function and therefore should not be affected. In contrast, the performance of a sales support program could be degraded. Depending on file size and system utilization, file distribution could take several minutes.

---

### Application Read Restrictions

An application should read only the prime version of a file having a file mode of Distribute At Close.

The reason for this restriction is important. Unless a TCLOSE had just completed, an application reading an image version of a file distributed at close could possibly receive obsolete data.

The importance of this restriction is seen in the case of an accounting totals file. This file, which is updated continually by a sales support application and distributed at close, needs the most current data, which is contained in the prime version.

In contrast, an application can read the image version of a per update file. If an application had to read the prime version of a per update file such as an item record file, then all price lookup data from store controllers would have to go across the LAN (MCF Network) and back. This could cause a performance problem.

---

### Spool Files on a LAN (MCF Network) System

If a terminal application writes to a prime version of a file that is on a store controller other than the one it is attached to, the record is sent on the token ring to the store controller, then over the LAN (MCF Network) to the other store controller.

If the remote store controller with the prime version of the file is disabled, the terminal application, with one exception, receives a write error because the record cannot be delivered over the LAN (MCF Network) to the disabled store controller.

The one exception is if the terminal application issues a WRITE MATRIX to a file that is owned by the file server. WRITE MATRIX is a 4680 BASIC language instruction that permits terminal applications to write string arrays to sequential files. See the *4680 BASIC: Language Reference* manual for details on the WRITE MATRIX instruction. Instead of generating a write error, the WRITE MATRIX record is saved in a

spool file created on the store controller that supports the terminal. These saved records are sent (despooled) to the intended file's prime version later, when the disabled store controller is returned to service.

This process is called WRITE MATRIX spooling, and is described by the example below, using a two-controller LAN system, with the prime version of the transaction log on the file server. The Alternate Master or Alternate File Server supports the TCC Network, and terminals on the network are using a sales application that is issuing a WRITE MATRIX at the end of each transaction.

If the file server becomes disabled, the Multiple Controller Feature on the alternate master or alternate file server creates a sequential spool file, `ADXFSSSF.SPL`, when it receives its first WRITE MATRIX from a terminal application.

As the alternate file server receives a normal WRITE MATRIX from each terminal on the network, it discards the WRITE MATRIX record, and a message is sent to the terminal telling it to send the record again as a different type of WRITE MATRIX record. This new WRITE MATRIX is the same as the original but has a 26-byte header containing the file name of the original target file. This header information is used later in despooling to send each record to the correct file.

The new WRITE MATRIX, with the header, is sent by the terminal applications addressed to the spool file now instead of the original file. The operating system receives the new WRITE MATRIX record and logs it to the spool file. Thus, the sequential spool file grows with each WRITE MATRIX.

When the file server is enabled again, the operating system determines that the file server is back up, and can now start logging directly to the prime version of the target file again. As each terminal sends in the next WRITE MATRIX, the header is stripped off, and the record is sent to the prime version on the file server. A message is sent to the terminal to begin sending each WRITE MATRIX to the prime version of the target file.

In the meantime, DDA starts despooling the spool file starting with the first record. The despooled records are sent to the prime version of the target file. Both the terminal application and DDA can be writing records to the same file. Therefore, the spooled records are intermixed with new records written by the terminals.

When the DDA comes to the end of the sequential spool file, it closes the file and starts issuing requests every two minutes to erase the spool file. Each erase command fails until all opens to the spool file have been closed. From the beginning, the system has counted the number of terminals that have been writing to the spool file, and decrements the count each time the system redirects a new terminal to the target file. When all terminals have been notified (the count goes to zero), the spool file is closed. At this point there are no opens to the spool file, DDA's erase command works, and the spool file disappears.

## Erasing the Spool File

To erase the spool file, all the terminals that wrote to the spool file are redirected to the target file. This redirection can only occur if each of those terminals continues to send in WRITE MATRIXes. If a terminal is inactive, or is turned off, the system is not able to reduce its spooling terminal count to zero and, therefore; it does not close the spool file. Consequently, the spool file cannot be erased by DDA. In this case, the spool file, even though it has been processed and is not being used, can not be erased right away.

**Note:** Leaving the spool file on disk does not cause problems. It does, however, require disk space.

## Forcing the Spool File to be Erased

One way to erase the spool file after all the transactions have been despoiled, but before the system program's terminal count has gone to zero, is to IPL the store controller that has the spool file. To the system program, the spool file is closed, and DDA is able to erase the file. There is no direct indication of this condition, however.

## Using Drive D: for the Spool File

Normally, the spool file is created automatically on the root directory of drive C. You can choose to have the spool file created at the root of drive D. This choice is made by using the User Logical Name table during configuration. The user logical file name should define ADXFSSSF to be D: ADXFSSSF.SPL. By creating a user logical file name that points to the drive D:, the system writes there rather than the default drive C:. Note that when you define the D:ADXFSSSF.SPL file, you receive a warning that you are defining an ADX file name and the system queries you. Click **Y** for yes.

## Saving Data That Cannot be Successfully Despoiled

In previous releases, any data found in the matrix write despool save file that either did not look like a proper record or could not be successfully despoiled was discarded. However, you now have an option to save data that cannot be successfully despoiled. You activate this option by defining a user logical file name for all controllers on the LAN. The logical file name is ADXFSSSF and can be either C:\ADXFSSSF.SAV or D:\ADXFSSSF.SAV. Any other value defaults to C:\ADXFSSSF.SAV.

Data found in the matrix write despool file that would previously have been discarded is now written to the save file specified, and no further action is taken. The save file is created if it does not exist. If the file does exist and there are no controls in the operating system to monitor the file's size, it is appended to.

It is your responsibility to maintain this file and to determine what, if anything, should be done with any data saved in the file. If the only data in this file is for file not found situations, you might be able to manually create the missing file and rename the save file back to the spool file name (that is, COPY ADXFSSSF ADXFSSSF). If you write a user program to process this file, you must open it shared; otherwise, new data can be lost if the despooler is unable to write to the save file.

The format of a valid record is a 26-byte file name field, which cannot be null terminated, then a 2-byte unsigned integer size, then the record to be written. When activated, the following additional W785 system message events can be generated: W785 MATRIX WRITE DESPOOLING EVENT HAS OCCURRED B4/S017/Exxx

- E021 Despooler unable to create save file
- E022 Despooler error occurred opening save file
- E023 Despooler error attempting to scan spool file
- E024 Despooler unable to obtain memory for save operation
- E025 Despooler unable to read data from spool file
- E026 Despooler unable to write save data to save file
- E027 Despooler data successfully written

All have unique data format 63 with the start offset and end offset referring to the portion of the spool file we are attempting to save. The despooler saves a maximum of 32 KB at a time so multiple successful writes or errors can follow W785 B4/S017/E011 or E012.

---

## Effects of Activating the Alternate File Server on Despooling

DDA tracks the log whether it is despoiled to the file server's prime version of the target file, or the (new) prime version on the alternate file server that is made acting file server. The only difference for the system is that the records are despoiled over the LAN (MCF Network) to the configured file server, or are despoiled locally (on the same store controller) to the alternate but acting file server. If the alternate file



server is made the acting file server, WRITE MATRIXes are now logged to the new prime version in the acting file server. The attempt to distribute the new records to the disabled file server is logged in the exception log as part of normal DDA activities for mirrored files. When the file server returns, DDA distributes the new records to the file server according to the exception log record entries to synchronize the mirrored files on both store controllers.

## Reactivating the Configured File Server

Before reactivating the file server, the acting file server has to be deactivated. For the period of time it takes to activate the configured file server, there is no acting file server. The system restarts the spooling process. The WRITE MATRIX records are written to the spool file until the configured file server is activated. Then DDA can distribute the spooled records over the LAN (MCF Network) to the appropriate target file on the configured file server.

---

## Record Sizes

The maximum TCC Network buffer size for sending WRITE MATRIX data to the store controller from the terminal application is 508 bytes. If the WRITE MATRIX data block is greater than 508 bytes, the block is divided into as many buffers as needed to comply with the 508-byte limit.

If the data received from the TCC Network by the store controller needs to be distributed to another store controller, and the total WRITE MATRIX data consists of more than one 508 buffer, a header count byte in the first buffer indicates this. A message tells the receiving store controller to expect the required number of buffers. When the whole WRITE MATRIX (consisting of more than one buffer) distribution is completed over the LAN (MCF Network), the sending store controller sends an unlock indicator as part of the last message to indicate that the data distribution is complete.

If a WRITE MATRIX sent by the terminal is less than 508 bytes, there is only one message sent per distribution. If the WRITE MATRIX length is over 508 bytes, then the number of LAN messages is the absolute value of  $(1 + n/508)$  where  $n$  is the size of the data block.

Thus the overhead of distributing a 509-byte data block (one over the limit) in terms of LAN messages required, is 2 to 1 for the 508 data block.

---

## Using the Application Program Interface (OS/2)

The application program interface between a store controller and an Operating System/2® (OS/2) PC enables file operations to be performed across the LAN using common application program statements. Application programs can be written to create, open, read, and write to files between two nodes on the LAN; thus providing a means of automating repeated file tasks.

To access a file on an OS/2 remote node, a 4690 application program must specify the remote node name. The complete file specification should follow this format:

*nodename::path:filename.ext*

The OS/2 computer name (the name designated during installation) should replace the *nodename*, and the shared file resource name should be used in the *path* position.

OS/2 application programs reference files on a store controller using the device name from the NET USE command. Before invoking an OS/2 application that opens or creates files on a store controller, the NET USE command should have been issued.

For a C-language program written to copy the General Sales Application transaction summary log from a store controller to an OS/2 machine, see “Example of a C/2 C-Language Program” on page 171. It shows that only standard C-language commands and library functions were used to perform the copy process.



The "Example of a 4680 BASIC Program" on page 173 also shows how to copy files between the operating system and OS/2. The program prompts the user for the source and destination file names prior to invoking the program statements that result in the file being copied. As noted in the program comments, this particular program is limited to copying files no larger than 32 767 bytes. This size restriction can be eliminated by using a more complex program that calculates the size of the file and determines an appropriate record size.

## Example of a C/2 C-Language Program

**Note:** The statements enclosed in a `/* */` are comments about the program.

```
/* This test program is designed to copy the          */
/* transaction log "EALTRANS" from 4690 to OS/2.      */
/* Prior to executing this program, virtual device E: */
/* must be defined by a NET USE statement as the      */
/* the ADX_SDT1 subdirectory of the store controller. */
/* This program can be compiled using the IBM C/2 compiler */
/* that is used with OS/2.                            */

#include <stdio.h>

main()
{
    FILE *FP1, *FP2;
    long length;
    char buf[512];

    /* Open the transaction log as a read-only, binary file at 4690 */
    if ((FP1 = fopen("e:altrans.dat", "rb")) == NULL) {
        perror("open of ealtrans.dat failed");
        abort();
    }

    /* Create a read/write binary file on the OS/2 machine. */
    if ((FP2 = fopen("trans.dat", "wb")) == NULL) {
        perror("open of transact.log failed");
        abort();
    }

    /* Copy the transaction summary log to the OS/2 file in blocks */
    /* of 512 bytes. */
    while ((length = fread(buf, 1, 512, FP1)) != 0)
        fwrite(buf, 1, length, FP2);

    /* Close the files and end the program. */
    if (fclose(FP1) != 0)
        perror("fclose of ealtrans.dat failed");
    if (fclose(FP2) != 0)
        perror("fclose of transact.log failed");

} /* End */
```

---

## Using the Application Program Interface (DOS)

The application program interface between the operating system and DOS enables file operations to be performed across the LAN using common application program statements. Application programs can be written to create, open, read, and write to files between two nodes on the LAN to provide a way to automate repeated file tasks.

To access a file on a remote DOS node, a 4690 application program must use a file specification that includes the node name of the remote computer. The complete file specification should follow this format:

*nodename::path:filename.ext*

The DOS computer name (the name designated by the NET START command) should be substituted for the *nodename*, and the shared resource short name should be used in the *path* position.

DOS application programs reference files on a store controller using the device name from the NET USE command. Before invoking a DOS application that opens or creates files on a store controller, the NET USE command must have been issued.

See “Example of a DOS C-Language Program” on page 172 for an example of a C-language program written to copy the General Sales Application transaction summary log from a store controller to a DOS PC. It shows that only standard C-language commands and library functions were used to perform the copy process.

“Example of a 4680 BASIC Program” on page 173 includes a 4680 BASIC program used to copy files between the two operating systems. The program prompts the user for the source and destination file names prior to invoking the program statements that result in the file being copied. As noted in the program comments, this particular program is limited to copying files no larger than 32 767 bytes. This size restriction can be eliminated by using a more complex program that calculates the size of the file and determines an appropriate record size.

## Example of a DOS C-Language Program

```
/* This test program is designed to copy a transaction log          */
/* "EALTRANS" from the 4690 to DOS. Prior to invoking the         */
/* program, virtual device E: must be defined by a NET USE        */
/* statement as the ADX_SDT1 subdirectory of the store            */
/* controller. This program was compiled using the                */
/* MetaWare High C compiler for use with DOS.                    */
                                                                    */

#include <stdio.h>
#include <stdlib.h>

void main()
{
    FILE *FP1, *FP2;
    int length;
    char buf[512];

    /* Open the transaction log as a read-only, binary file       */
    /* at the store controller.                                    */
    if ((FP1 = fopen("e:ealtrans.dat", "rb")) == NULL) {
        perror("open of ealtrans.dat failed");
        abort();
    }

    /* Create a read/write, binary file on the DOS machine.       */
    if ((FP2 = fopen("transact.log", "wb")) == NULL) {
        perror("open of transact.log failed");
        abort();
    }

    /* Copy the transaction summary log to the DOS file in        */
    /* blocks of 512 bytes                                         */
    while ((length = fread(buf, 1, 512, FP1)) != 0)
        fwrite(buf, 1, length, FP2);

    /* Close the files and end the program.                       */
    if (fclose(FP1) != 0)
        perror("close of ealtrans.dat failed");
}
```

```

    if (fclose(FP2) != 0)
        perror("close of transact.log failed");
} /* END */

```

## Example of a 4680 BASIC Program

This BASIC program copies files between two nodes on the LAN. The program prompts the user for the source and destination file names before invoking the program statements that result in the file being copied. As noted in the program comments, this particular program is limited to copying files no larger than 32 767 bytes. This size restriction can be eliminated by using a more complex program that calculates the size of the file and determines an appropriate record size.

```

REM   This program allows 4690, DOS, or OS/2 on the LAN to copy
REM   files of up to 32,767 bytes, to itself or other 4690, DOS, or OS/2
REM   on the LAN.
REM
REM   Provide the sending and receiving store controllers netnames,
REM   directories, and filenames when prompted.
REM
REM   For example: Sending Controller      DOMAIN::CDISK:FILE1
REM                  Receiving Controller  ADXLCCN::C:NEWFILE1
REM
REM   The previous example would cause FILE1 from the C
REM   disk subdirectory of the store controller, DOMAIN1,
REM   to be copied to the store controller, ADXLCCN.
REM   When FILE1 is placed in the root directory, it
REM   is renamed as NEWFILE1.
REM
integer*4 fsize%,rnum%
on error goto 1000
100  CLEARS
    print " ***** 4690 - OS/2 - DOS Connectivity Test Program ***** "
    print " "
    print "      TO EXIT THIS PROGRAM JUST PRESS THE ENTER KEY "
    print "      WHEN PROMPTED FOR THE SENDING OR RECEIVING CONTROLLERS"
    print " "
    input " ENTER THE SENDING CONTROLLER NAME AND PATH "; sender$
    print " "
    if sender$ = " " then goto 2000          ! EXIT THE PROGRAM
    input " ENTER THE RECEIVING CONTROLLER NAME AND PATH "; receiver$
    print " "
    print " "
    if receiver$ = " " then goto 2000        ! EXIT THE PROGRAM
    print " "
    ios1% = 1          ! ESTABLISH I/O SESSION NUMBER FOR SENDER
    ios2% = 2          ! ESTABLISH I/O SESSION NUMBER FOR RECEIVER
    rnum% = 1
    fsize% = size(sender$)      ! DETERMINE THE SIZE OF THE FILE
    if fsize% > 32767 then 1500 ! EXIT PROGRAM IF FILE IS GREATER THAN MAX
    if fsize% > 0 then 250      ! FILE IS VALID KEEP PROCESSING
    print "THIS PROGRAM COPIES FILES WITH A SIZE RANGE OF 1 - 32767 BYTES "
    print "THE FILE YOU SPECIFIED TO BE COPIED IS EMPTY "
    goto 2000                ! EXIT PROGRAM BECAUSE INPUT FILE IS EMPTY
250  print " THERE ARE"; fsize%; "BYTES IN THE INPUT FILE "
    numchars$ = "c"+str$(fsize%) ! CONVERT SIZE TO CHARACTER FOR READ/WRITE
    open sender$ rec1 fsize% as ios1% ! OPEN INPUT FILE
    openin% = 1          ! SUCCESSFUL OPEN INDICATOR
    create receiver$ rec1 fsize% as ios2% ! CREATE A NEW OUTPUT FILE
    openout% = 1         ! SUCCESSFUL OPEN INDICATOR
    read from numchars$; #ios1%,rnum%; data1$ ! READ THE DATA FROM SENDER
    write from numchars$; #ios2%,rnum%; data1$ ! WRITE THE DATA TO RECEIVER
    print " "
    print " COPY COMPLETE "
    close ios1%          ! CLOSE INPUT FILE

```

```

        close ios2%           ! CLOSE OUTPUT FILE
        wait;5000             ! PAUSE
        goto 100              ! GO BACK TO THE BEGINNING
1000  print " AN ERROR HAS OCCURRED "
        print " CHECK THE DATA YOU INPUT FOR THE SENDER/RECEIVER"
        wait;5000             ! PAUSE SO THAT MESSAGE CAN BE READ
        if openin% <= 0 then 1200 ! DETERMINE IF INPUT FILE HAD BEEN OPENED
        close ios1%           ! CLOSE INPUT FILE
        if openout% <= 0 then 1200 ! DETERMINE IF OUTPUT FILE HAD BEEN OPENED
        close ios2%           ! CLOSE OUTPUT FILE
1200  resume 100              ! GO BACK TO THE BEGINNING
1500  print " "
        print "FILES > 32767 BYTES CANNOT BE PROCESSED BY THIS PROGRAM"
        print "THE SIZE OF THE FILE YOU SPECIFIED IS "; FSIZE%; "BYTES"
2000  Print " "
        Print " "
        Print "*****      THE PROGRAM IS CANCELED      *****"
        stop
        end

```

---

## Part 2. Utilities



---

## Chapter 6. Using the Keyed File Utility

The operating system provides a Keyed File Utility (KFU) to enable you to create and manage keyed files. You can start the utility from the SYSTEM MAIN MENU panel, from Command Mode, or from the host processor.

When you start the KFU from the SYSTEM MAIN MENU, you are using it in an operator-interactive mode. When you start the utility from a batch file, you provide the necessary input on the command statement that starts the utility. When you start the utility from the host processor, you provide the necessary input through Host Command Processor (HCP) commands. See the *4690 OS: Communications Programming Reference* for information on using the KFU from the host processor.

To learn more about the characteristics of keyed files, refer to the “Internal Processes of Keyed Files” on page 185.

---

### Accessing the Keyed File Utility

The KFU is accessed using menu panels. From the SYSTEM MAIN MENU, select **File Utilities**. When the FILE UTILITIES panel appears, select the **Keyed File Utilities** option. When you select it, the ORGANIZE KEYED FILES panel appears.

This panel, and the ones that follow it, guide you in creating keyed files or checking the performance of existing keyed files. You can also create keyed files from a BASIC application program using the CREATE POSFILE statement.

---

### Using the Keyed File Utility from the Host

The following steps explain how to invoke KFU from the host:

1. Use the ADCS CREATE command.
2. You can create an ADXCSKPF.DAT file at the host, send it to the 4690, and start KFU as a background program from the host. The KFU looks for the ADXCSKPF.DAT file, uses it to get the required parameters, and performs the operation you specified.

These steps are:

- a. Create the ADXCSKPF.DAT file at the host. Each record in the file is a separate KFU command. Each record should have the parameters *a* through *m* starting with *a*. Do not include the ADXCSK0L command at the beginning of each record.
- b. Be sure the logical names table on the 4690 includes the logical name for ADXCSKPF.
- c. Send the ADXCSKPF.DAT file to the store controller.
- d. Start KFU as a background program on the 4690 from the host. See the *4690 OS: Communications Programming Reference* for more information.
- e. The KFU puts “X\*X\*X\*X ” in place of each record in the ADXCSKPF.DAT file that completes successfully. If a record does not complete successfully, the record remains untouched. By pulling this file back to the host at the end of the job, you can determine the completion status. You can resend this file with the completion status records to retry the records that did not complete successfully because KFU ignores the records containing “X\*X\*X\*X ”.

If a record in the ADXCSKPF.DAT file is less than eight characters long, KFU writes only a left substring of “X\*X\*X\*X ” at the successful completion of the record. The length of the substring is the length of the record. For example, the function to cancel a checkpoint needs only one parameter (the character 8). KFU writes X in place of 8 to indicate the successful completion of the function.

## Creating a Keyed File

An existing direct file is used as input to create a keyed file. Before creating the keyed file, you must determine values for the following variable parameters:

- Size of the keyed file

The size of a keyed file does not change. When you create a keyed file, specify more than the actual number of records you intend to place in the file. If you anticipate that the actual number of records in the file will increase, specify your best guess for the maximum number of records the file will ever hold. After you have determined the maximum number of records for the file, increase the number by 25% to allow for adequate chaining space.

- Hashing algorithm

See “Hashing Algorithms” on page 183.

- Randomizing divisor

Selection of an efficient randomizing divisor is important in minimizing the number of chained records in a keyed file. Chained records require more disk read operations for retrieval than do home records. Therefore, minimizing chained records maximizes performance when accessing the file by key.

Before selecting a randomizing divisor, calculate the number of blocks in the file as follows:

Records per block = 508 / record size  
Total blocks = total records / records per block

The randomizing divisor must be less than the total blocks in the file by one or more. In general, prime numbers are best. A good randomizing divisor might be the highest prime number that is less than the number of blocks in the keyed file. The efficiency of a randomizing divisor can be tested by requesting the Chaining Statistics from a Direct File option on the KEYED FILE MANAGEMENT panel. You can specify a randomizing divisor and hashing algorithm to be analyzed. The direct file is scanned to report the chaining, distribution, and packing. A chained record percentage greater than 10% is normally considered too inefficient.

When you have selected your file size, randomizing divisor, and hashing algorithm, request the Create Keyed File option on the KEYED FILE MANAGEMENT panel. You are prompted to enter all of the variables needed to create the keyed file. The KFU begins writing records to the keyed file.

The KFU builds the keyed file in three phases:

1. The utility writes the home records to the keyed file, and places the channel records in a work file.
2. The utility sorts the work file.
3. The utility writes chained records to the keyed file.

Information messages appear on your panel as the keyed file is being created. The system displays the current phase and a count of the records that the utility has written to the keyed file. Checkpoints are taken by the KFU at 20-minute intervals. When creation of the keyed file is complete, the utility asks if you would like to erase the input file. If you do not need the input file for any other purpose, erase it.

If the keyed file creation process is interrupted, you can restart the process from the last checkpoint. When you start the KFU after creation of a keyed file has been interrupted, the CHECKPOINT RESTART menu appears. You can restart creation of the keyed file or cancel the checkpoint.

## Creating a Direct File

The KFU enables you to create a direct file using a keyed file as input. You specify this option on the KEYED FILE MANAGEMENT panel and fill in the name of the keyed file to be used as input. When the utility creates a direct file from a keyed file, it reverses the process described previously. It creates a direct file of the records found in the keyed file by reading the file sequentially and writing the records to a direct file as it reads them.



## Reporting Keyed File Statistics

The KFU can display statistics on each keyed file created. These statistics include the number of reads, writes, deletes, opens, and closes done on files. They also include information on the number of sectors and chains the file contains. You can examine these statistics and reset them to zero as required.

## Creating an Empty Keyed File

The KFU can create an empty file. Terminal and store controller applications can add records to an empty file as needed.

## Chaining Statistics from a Direct File

The KFU can test the efficiency of randomizing divisors and hashing algorithms before you create a keyed file. The direct file is scanned to report the percentage of chained records and the distribution.

## Rebuilding a Keyed File

The KFU enables you to rebuild a keyed file. You need to recreate a keyed file if you want to change its options. To do this, first convert the keyed file to a direct file by selecting the Create a Direct File option on the KEYED FILE MANAGEMENT panel.

After you convert your file, you can access it and make the necessary changes. Then recreate the keyed file using the Create a Keyed File option with the direct file as the input.

---

## Using the Keyed File Utility in a Batch File

If you have several keyed files to create, you can set up a batch file to invoke the KFU for each file to be created.

The following items can be performed in a batch file with the Keyed File Utility.

- Create a keyed file from a direct file.
- Create a direct file from a keyed file.
- Create an empty file.
- Produce statistics for a keyed file.
- Report chaining statistics from a direct file.
- Restart from a checkpoint.
- Restart from the last checkpoint if the checkpoint exists for the specified keyed file. If a checkpoint does not exist for the specified keyed file, delete the checkpoint and create the specified keyed file from a direct file.
- Cancel the checkpoint and erase the checkpoint records.
- Cancel the checkpoint and create a keyed file from a direct file.

The KFU does not modify the batch file as a result of executing it.

## Creating a Keyed File from a Direct File

Use the following command line in the batch file:

### Format

```
ADXCSK0L a b c d e f g h i j k l m n o p
```

where:

- a**      Function request. Valid values are:
- 1**      Create a keyed file from a direct file.

- 7 Restart from the last checkpoint if the checkpoint exists for the specified keyed file. If a checkpoint does not exist for the specified keyed file, delete the checkpoint, and create the specified keyed from a direct file.
- 9 Cancel the checkpoint and create a keyed file.
- b Report output device:
  - 2 Printer output
  - 3 File output
- c Disposition of direct file after processing is finished:
  - N Do not erase the input file.
  - Y Erase the input file.
- d Disposition of an existing version of the keyed file:
  - N Do not erase an existing version of the keyed file.
  - Y Erase an existing version of the keyed file.
- e Name of direct file (up to 35 characters).
- f Name of keyed file (up to 35 characters).
- g Record length of a direct file (range of 1 to 508).
- h Record length of keyed file (range of 1 to 508).
- i Length of key ( $\leq$  record length).
- j Number of records to be allocated on disk. (This should be the maximum number of records anticipated plus a minimum of 25%.)
- k Randomizing divisor. If  $k = 0$  is entered, a system generated randomizing divisor is used.
- l Chaining threshold.
- m File attributes:
  - 1 Local
  - 2 Mirrored at update
  - 3 Mirrored at close
  - 4 Compound at update
  - 5 Compound at close
- n Hashing algorithm. (This optional parameter can be used to override any hashing algorithm specified using logical names. It can be 0, 1, or 2. See "Hashing Algorithms" on page 183.)
- o Maximum in memory table buffers. (This optional parameter can be used to limit the number of 32-KB buffers that can be allocated.)
- p Minimum in memory table buffers. (This optional parameter can be used to specify the minimum number of 32-KB buffers to be allocated.)

**Note:** Parameters  $n$ ,  $o$ , and  $p$  are optional. They are positional parameters. If one is specified, all of the preceding parameters must be specified.

F1 is not accepted if the checkpoint file exists.

## Creating a Direct File from a Keyed File

Use the following command line in the batch file:

**Format**

```
ADXCSK0L 2 b c d e f m
```

where:

- b** Report output device:
  - 2** Printer output
  - 3** File output
- c** Disposition of keyed file after processing is finished:
  - N** Do not erase keyed file when finished.
  - Y** Erase keyed file when finished.
- d** Disposition of an existing version of the direct file:
  - N** Do not erase an existing version of the direct file.
  - Y** Erase an existing version of the direct file.
- e** Name of keyed file (up to 35 characters).
- f** Name of direct file (up to 35 characters).
- m** File attributes:
  - 1** Local
  - 2** Mirrored at update
  - 3** Mirrored at close
  - 4** Compound at update
  - 5** Compound at close

## Creating an Empty File (Batch Method)

Use the following command line in the batch file:

**Format**

```
ADXCSK0L 4 b c d e f g h i j k l m n
```

where:

- b** Report output device:
  - 2** Printer output
  - 3** File output
- c** Y or N (used only as a placeholder).
- d** Disposition of an existing version of the keyed file:
  - N** Do not erase an existing version of the keyed file.
  - Y** Erase an existing version of the keyed file.
- e** Dummy name (used only as a placeholder).
- f** Name of keyed file (up to 35 characters).
- g** Record length (used only as a placeholder).
- h** Record length of keyed file (range of 1 to 508).
- i** Length of key (<= record length).
- j** Number of records to be allocated on disk.

- k** Randomizing divisor. If  $k=0$ , a system generated randomizing divisor is used.
- l** Chaining threshold.
- m** File attributes:
  - 1** Local
  - 2** Mirrored at update
  - 3** Mirrored at close
  - 4** Compound at update
  - 5** Compound at close
- n** Hashing algorithm. (This optional parameter can be used to override any hashing algorithm specified using logical names. It can be 0, 1, or 2. See “Hashing Algorithms” on page 183.)

## Reporting Keyed File Statistics

Use the following command line in the batch file:

**Format**  
 ADXCSK0L 3 *b c d e*

where:

- b** Report output device:
  - 2** Printer output
  - 3** File output
- c** Y (not used)
- d** Disposition of statistics after producing the report:
  - N** Do not clear statistics after producing report.
  - Y** Clear statistics after producing report.
- e** Name of keyed file (up to 35 characters).

## Reporting Chaining Statistics from a Direct File

Use the following command line in the batch file:

**Format**  
 ADXCSK0L 5 *b c g i j k [n] [o] [p]*

where:

**Note:** Parameters *n*, *o*, and *p* are optional. They are positional parameters. If one is specified, all of the preceding parameters must be specified.

- b** Report output device:
  - 2** Printer output
  - 3** File output
- c** Name of direct file (up to 35 characters).
- g** Record length of keyed file (1 to 508).
- i** Length of key ( $\leq$  record length).

- j** Number of records to be allocated in the keyed file.
- k** Randomizing divisor. If  $k=0$ , a system generated randomizing divisor is used.
- n** Hashing algorithm. (Must be 0, 1, or 2. See “Hashing Algorithms” on page 183.)
- o** Maximum in memory table buffers. (This optional parameter can be used to limit the number of 32-KB buffers that can be allocated.)
- p** Minimum in memory table buffers. (This optional parameter can be used to specify the minimum number of 32-KB buffers to be allocated.)

## Restarting from a Checkpoint

Use the following command line in the batch file:

<b>Format</b> ADXCSK0L 6
-----------------------------

## Canceling (Erasing) a Checkpoint

Use the following command line in the batch file:

<b>Format</b> ADXCSK0L 8
-----------------------------

---

## Keyed File Utility Working Files

The following list shows the logical names of working files created or updated by the KFU:

<b>ADXCSKPF</b>	Parameter file created at the host
<b>ADXCSKOF</b>	Report file
<b>ADXCSKTF</b>	Temporary file used when the input file has the same name as the output file
<b>ADXCSKCP</b>	Checkpoint file
<b>ADXCSKCK</b>	Temporary checkpoint file
<b>ADXCSKST</b>	Sector table file
<b>ADXCSKSS</b>	Copy of the sector table file used for checkpoint restart
<b>ADXCSKCF</b>	Temporary report work file
<b>ADXCSKWF</b>	Work file for chained records
<b>ADXCSKAF</b>	Work file for sort
<b>ADXCSKBF</b>	Work file for sort

---

## Hashing Algorithms

The operating system provides a default hashing algorithm for all keyed files: the Folding algorithm. It also provides the Exclusive OR (XOR) rotation hashing algorithm and the polynomial hashing algorithm for use on large keyed files. Using the XOR rotation hashing algorithm or the polynomial hashing algorithm on large keyed files results in fewer chained records and improved performance in file build and file access.

The polynomial hashing algorithm improves record distribution when keys contain repeated numeric patterns. A key that has ASCII characters is an example of a key with repeated numeric patterns.

Hashing algorithms other than the default should be chosen for files having more than 40-KB records and a randomizing divisor greater than 6000. An item record file is an example of a file whose performance might be improved through use of a hashing algorithm other than the default.

**Attention:** The XOR rotation hashing algorithm and the polynomial hashing algorithm cannot be used on the Item Movement File of the 4680 or 4680-4690 Supermarket Application.

You can select the hashing algorithm when a keyed file is created and specified through logical names. The logical names are entered using the User Logical File Names option on the CONTROLLER CONFIGURATION panel. You must define the logical name at the store controller (master store controller or file server) where the file is created. Perform the following steps:

**Note:** If you are using the Folding hashing algorithm, you do not need to perform this procedure.

1. From the SYSTEM MAIN MENU, select the Installation and Update Aids option.  
The INSTALLATION AND UPDATE AIDS panel appears.
2. Select Change Configuration.  
The CHANGE CONFIGURATION panel appears.
3. Select Controller Configuration, and then select User Logical File Names.
4. Define the logical name for the file. The logical name is the file name with an H appended, for example, EALITEMRH for the file EALITEMR.DAT. The name is the same as it appears in the disk directory but without the file extension.

The User Logical File Names panel is displayed. Select the Define a Logical File Name option, and type the logical name (the file name with the H appended). The Define Logical File Names panel appears. Enter either **0**, **1**, or **2** on this panel to define the appropriate hashing algorithm for use with the file:

- 0** Folding algorithm
- 1** Exclusive OR (XOR) rotation hashing algorithm
- 2** Polynomial hashing algorithm

For example, to select the XOR rotation hashing algorithm for file EALITEMR.DAT, the logical name EALITEMRH = 1.

When the system creates a file, it searches the logical names table in the following order:

- File name with H appended
  - If the above file name is not found, it searches the system default hashing algorithm name ADXHASH.
  - If neither logical file name is found, the system selects the default hashing algorithm.
5. After defining the logical name, return to the CONTROLLER CONFIGURATION panel and select the Activate Configuration option.
  6. When you activate the configuration, IPL the store controller to load the new definitions. You can verify the definitions as described in "Verifying Definitions" on page 185.
  7. If the logical name shows the correct hashing algorithm, create the keyed file. Use either a 4680 BASIC program or the KFU.

**Note:** If you use the KFU to create the keyed file, you can override the hashing algorithm selection made using logical names.

## Specifying Algorithms for Files

If you want to know which algorithm has been specified for a particular file, use the following procedure:

1. On the SYSTEM MAIN MENU, select the File Utilities option.
2. When the next panel appears, select the Keyed File Utilities option.
3. When the next panel appears, select the Performance Statistics of a Keyed File option.

This function of the KFU produces a report that includes the hashing algorithm specified for the particular file.

If you want to change the hashing algorithm on a keyed file, use the KFU to create a direct file from the keyed file. Then erase the keyed file. Create the keyed file from the direct file by specifying the appropriate hashing algorithm.

You can define the system default hashing algorithm using the logical name ADXHASH. If you define logical name ADXHASH as 1, all keyed files created use the XOR rotation hashing algorithm. If you define the logical name ADXHASH as 2, all keyed files created use the polynomial hashing algorithm. The KFU has an optional parameter to override the system default when creating a keyed file from a direct file.

## Verifying Definitions

To verify new definitions that have been activated, select the Command Mode option from the SYSTEM MAIN MENU. When the prompt appears, enter **DEFINE -S-N** *logical name*. The system searches the logical names table for the logical name that you entered.

---

## Internal Processes of Keyed Files

This section contains additional information on the internal processes of keyed files. It explains the algorithms used in keyed files to randomize record keys. It explains how the record chaining is used. This section also contains a layout of the control fields in the first block of a keyed file.

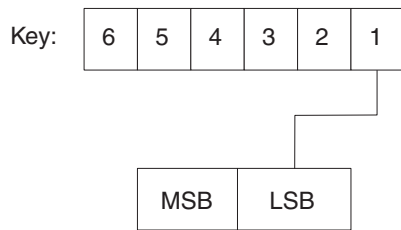
## Folding Algorithm

The Folding algorithm randomizes the distribution of keyed records in the data set. Randomizing transforms record keys into expected block addresses relative to the start of the data set. The procedure is divided into two distinct parts: *key folding* and *randomizing*.

### Key Folding

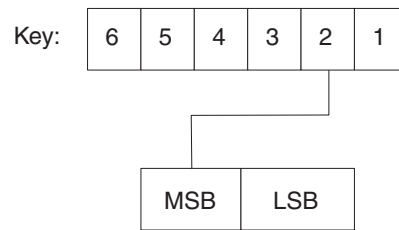
Folding is a logical accumulation of the bytes of the key into two accumulation bytes; these bytes are initialized to zero. Alternate key bytes are folded into accumulators with the Exclusive OR (XOR), starting with the low-order key byte into the low-order accumulator and the next byte of the key into the high-order accumulator. This process continues, working through the key from low-order to high-order (right to left), until the key is entirely processed. The result is a two-byte value derived from the entire key. See Figure 8 on page 186 for more information.

Step 1:

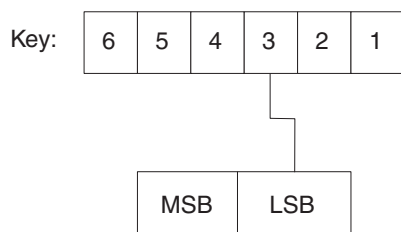


--Exclusive OR--  
-Folded Key-

Step 2:

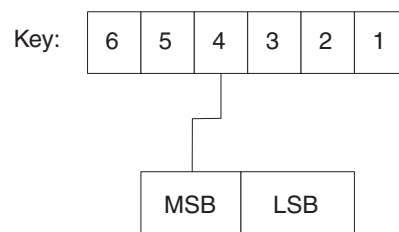


Step 3:

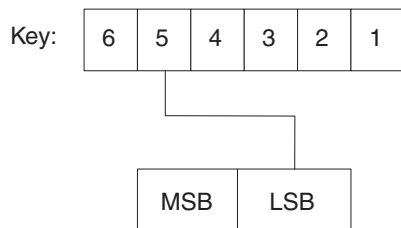


--Exclusive OR--  
-Folded Key-

Step 4:



Step 5:



--Exclusive OR--  
-Folded Key-

Step 6:

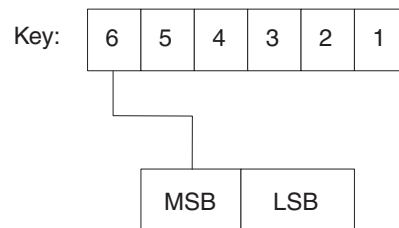


Figure 8. Folding Algorithm

#### Notes:

1. MSB denotes the most significant byte.
2. LSB denotes the least significant byte.

### Randomizing

A randomizing divisor divides the folded key. The divisor must be less than the number of blocks allocated for the keyed data set. The remainder from the division is the expected relative block number. The choice of the randomizing divisor and record keys affects the distribution of records in the data set. You must choose these variables so that the data set contains a uniform distribution of records. While no records randomize greater than the randomizing divisor, these blocks are used for overflow records if any additional blocks are needed and available.

Randomization is performed by dividing the two-byte folded key (MSBLSB) by the randomizing divisor. The resulting remainder is the relative block number in the keyed file that contains the keyed logical record or an overflow chain pointer to the block containing the record. If the remainder is equal to zero, the relative block number is equal to the randomizing divisor.



## The XOR Rotation Hashing Algorithm

The XOR rotation hashing algorithm expands the length of the folded key produced by the key folding step of the folding algorithm. It also rotates the bits in a random way to produce less record chaining. The algorithm performs the following operations:

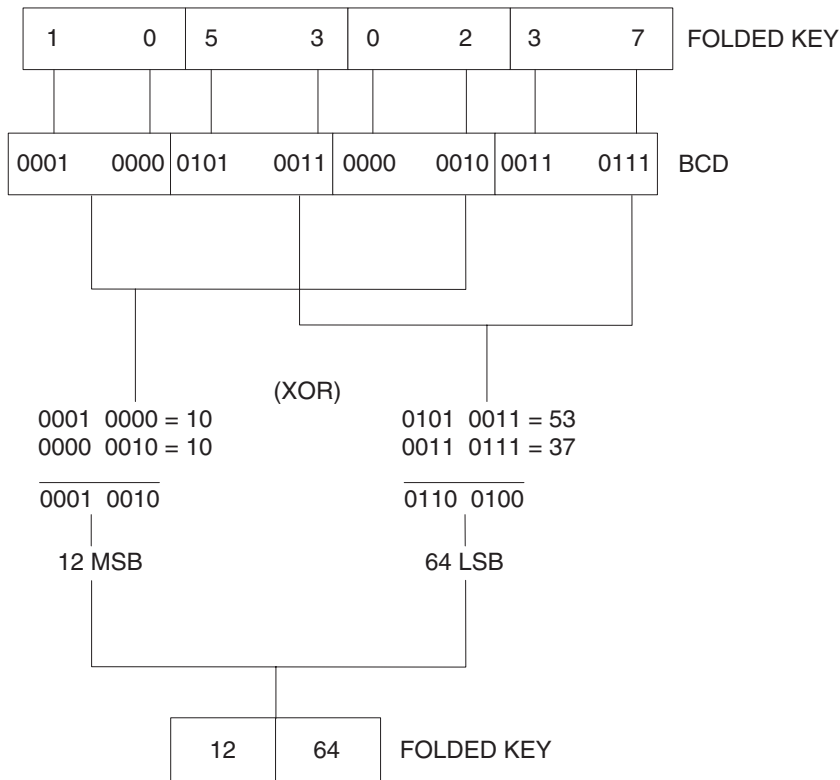
1. Calculates the number of bit positions of rotation to be done after folding (a number from 0 to 15).
  - a. Adds each digit (or half-byte) to a sum as the key is scanned from right to left during folding. Prior to this addition, the sum is multiplied by 10. This effectively reverses the digits of a packed decimal key and converts to binary. To prevent a 32-bit overflow, the conversion is ended if the sum reaches 214 748 364. During the scan, the summing process does not begin until a nonzero byte is encountered. The sum does not include null bytes that are right-justified in the key.
  - b. Divides the resulting sum by 31 (a prime number). The remainder is halved to produce the bit rotation count from 0 to 15.
2. Rotates the bits in the folded key the number of positions calculated in Step 1. The bits are rotated to the left. Bits shifted out of the high-order position move to the low-order position.
3. Combines the rotated folded key with the original folded key to form a 32-bit number. The rotated folded key becomes the high-order word of the 32-bit number. The original folded key becomes the low-order word of the 32-bit number.
4. Rotates the bits in the 32-bit number to the left the number of positions calculated in Step 2 on page 187.
5. Zeroes the high-order bit of the 32-bit number to prevent a negative result.
6. Divides the 32-bit number by the randomizing divisor. The remainder becomes the relative block number of the record.

## Example of the XOR Rotation Hashing Algorithm

Assume a 4-byte key of eight packed decimal digits shown here.

10	53	02	37
----	----	----	----

The randomizing divisor is 49997.



**Note:**

BCD denotes "binary code decimal"

MSB denotes "most-significant byte"

LSB denotes "least-significant byte"

XOR denotes "Exclusive OR" explained below:

Bit 1	Bit 2	Result
0	0	0
0	1	1
1	0	1
1	1	0

The folding algorithm (see Figure 8 on page 186) is performed to produce this folded key.

12	64
----	----

As it computes the folded key, the algorithm calculates the bit rotation count as follows:

1. Converts the digits to binary in reverse order and divides by 31.  $73203501 / 31 = 2361403$  with a remainder of 8.
2. Halves the remainder to produce the bit rotation count.  $8 / 2 = 4$ .

Here is the folded key  
expressed in bit notation.

0001 0010	0110 0100
-----------	-----------

Here the key has been rotated  
4 positions to the left.

0010 0110	0100 0001
-----------	-----------

0010 0110	0100 0001	0001 0010	0110 0100
-----------	-----------	-----------	-----------

The rotated key is concatenated with the original folded key  
to form the 32-bit number shown above.  
The 32-bit number is rotated 4 positions to the left.

0001 0010	0101 0011	0000 0010	0011 0111
-----------	-----------	-----------	-----------

The hexadecimal representation of the result is 64112642.  
The decimal value is 1,678,845,506.  
The 32-bit number is divided by the randomizing divisor, 49997, giving  
33578 with a remainder of 46240.

The home block of the key is remainder, 46240.

## Polynomial Hashing Algorithm

The polynomial hashing algorithm uses the same technique that generates the Frame Check Sequenced (FCS) field for SDLC transmission frame. This algorithm is for users who choose to generate the FCS without using the Hashing Utility, KFU.

The algorithm performs the following steps:

1. Generate a 16-bit FCS number from the key of a record.
2. Reverse the key byte by byte.
3. Generate another 16-bit FCS number from the reversed key.
4. Concatenate the 16-bit numbers from Step 1 and Step 3 to make a 32-bit number. The high-order 16 bits are from Step 3 and the low-order 16 bits are from Step 1.
5. Turn off the high-order bit of the 32-bit number from Step 4.
6. Divide this 32-bit number by the randomizing divisor. The remainder becomes the relative block of the record.

## Example 1

The following example illustrates the polynomial hashing algorithm. You can use the following application to generate a keyed file without using the KFU.

Use the same 4-byte key used in the XOR rotation hashing algorithm example, 10530237. Assume the randomizing divisor is 49997.

Key	16-Bit FCS Number	Hexadecimal Equivalent	Bit Notation
10530237	9588	2574	0010 0101 0111 0100
Reversed Key	16-Bit FCS Number	Hexadecimal Equivalent	Bit Notation
37025310	48043	BBAB	1011 1011 1010 1011

**Concatenation of these two numbers equals:**

1011 1011 1010 1011      0010 0101 0111 0100

**When you turn off the high-order bit, the number equals:**

0011 1011 1010 1011      0010 0101 0111 0100

**The hexadecimal representation of the result equals:**

3BAB2574

**The decimal value equals:**

1 001 071 988

**Divide the decimal value (1 001 071 988) by the randomizing divisor (49997) giving:**

20022 Remainder = 32054

**The home block of the key (10530237) equals:**

Remainder = 32054 Hexadecimal Value = 7D36

## Example 2

Another example uses the 4-byte key 00008998. Again, assume the randomizing divisor is 49997.

Key	16-Bit FCS Number	Hexadecimal Equivalent	Bit Notation
00008998	49099	BFCB	1011 1111 1100 1011
Reversed Key	16-Bit FCS Number	Hexadecimal Equivalent	Bit Notation
98890000	26427	673B	0110 0111 0011 1011

**Concatenation of these two numbers equals:**

0110 0111 0011 1011      1011 1111 1100 1011

**When you turn off the high-order bit, the number equals:**

0110 0111 0011 1011      1011 1111 1100 1011

**The hexadecimal representation of the result equals:**

673BBFCB

**The decimal value equals:**

1 731 968 971

**Divide the decimal value (1 731 968 971) by the randomizing divisor (49997) giving:**

34641 Remainder = 22894

**The home block of the key (00008998) equals:**

Remainder = 22894 Hexadecimal Value = 596E

## Example C-language Program

This example program is written in C language. It is used to generate a 16-bit FCS number.

```
unsigned short crcgen (data, length)
char *data;          /* address of the key */
int length;          /* key length */
{
    unsigned char crcbyte1, crcbyte2, r1;
    char *ptr;
    int i;
    union byte_to_word
    {
        char bytes[2];
        unsigned short ret;
    } return_val;

    ptr = data;
    crcbyte1 = 0xFF
    crcbyte2 = 0xFF

    for (i=0; i<length; i++)
    {
        r1 = *ptr++ ^ crcbyte2;
        r1 = (r1 << 4) ^ r1;
        crcbyte2 = (r1 << 4) | (r1 >> 4);
        crcbyte2 = (crcbyte2 & 0x0F) ^ crcbyte1;
        crcbyte1 = r1;
        r1 = (r1 << 3) | (r1 >> 5);
        crcbyte2 = crcbyte2 ^ (r1 & 0xF8);
        crcbyte1 = crcbyte1 ^ (r1 & 0x07);
    }

    return_val.bytes[0] = ~crcbyte2;
    return_val.bytes[1] = ~crcbyte1;
    return(return_val.ret);
}
```

The following table shows the output from the example program when used with the input data shown. The I/O is two bytes and hexadecimal form.

Input	Output
1234	DEC1
1111	8206
4143	2066
8181	OFD2
8998	4C52

## Record Chaining

Each block has two chain pointers, each two bytes long. One pointer points forward, the other, backward. The system uses these pointers to chain together records that overflow one block into another. The next block that the overflow data is put into cannot be the next sequential sector on the disk. By chaining them together using pointers, the system knows which home block the data is related to.

The block the overflow data is related to is called a *home block*. If the system tries to add a record to a home block that is already full of records, it checks to see if an overflow chain exists for that block. If one does, the system checks it for available space. If the system finds the available space, it adds the record to the block with the available space. If there is no available space in that overflow chain, the system locates another block without overflow records, and adds this block to the end of that block's overflow chain.

If no overflow chain exists for the original home block, the system creates one. It scans the keyed file for a block with sufficient room for the record. The block cannot already contain other overflow records. The record is added to this block; this block is added to the record's home block overflow chain.

**Note:** Overflow chains **do not** contain overflow records from more than one home block.

When creating a keyed file, the system sets a value for a number called the *chaining threshold*. The system logs a message in the system log when adding records and an overflow chain is encountered that is longer than the threshold value. This message indicates that either the keyed data storage is getting too full or that the file is poorly randomized.

Figure 9 on page 192 through Figure 12 on page 193 illustrate space management with and without overflow chains.

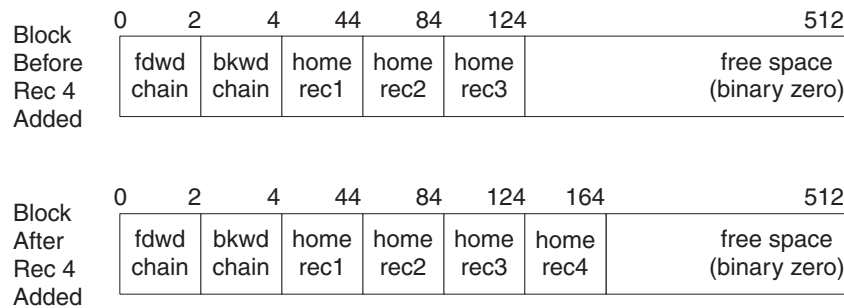


Figure 9. Example of Home Record Add without Overflow Present

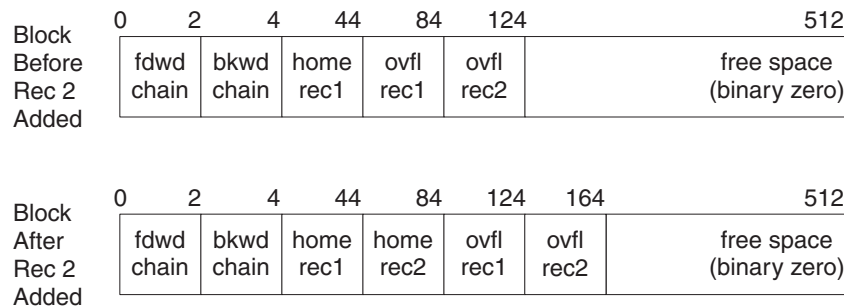
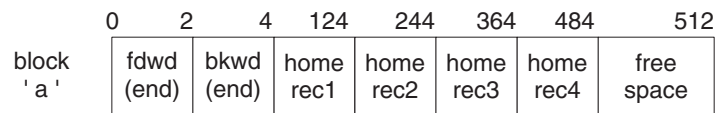


Figure 10. Example of Home Record Add with Overflow Present

Data set before record five added to block 'a'.



Data set after overflow block allocated and record five added. Record five becomes the first overflow chain for block 'a'.

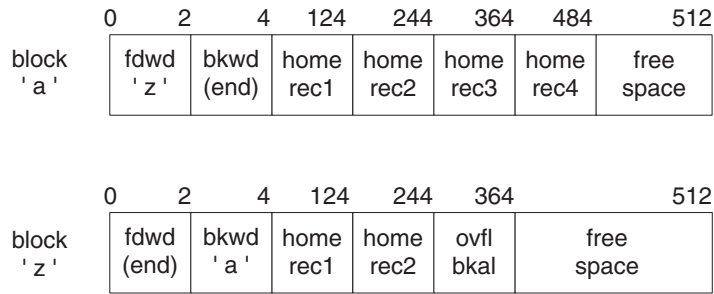
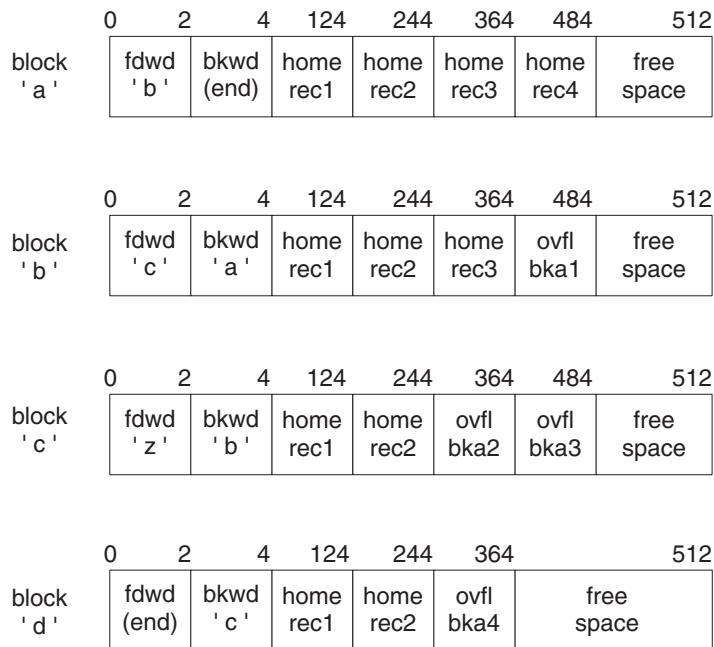


Figure 11. Example of Adding a Record to a Full Home Block Creating Overflow Chain

**Note:** The same process is performed if the last block in the overflow chain is full. Space is found in another block not containing any overflow records, and this block is added to the current overflow chain.

Data set before record three added to block 'c'



Data set after record three added to block 'c'

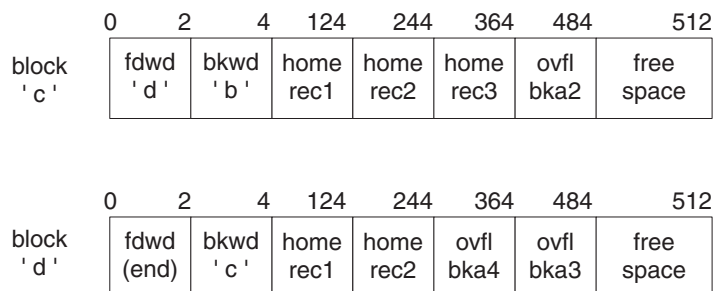


Figure 12. Example of Home Record Add with Overflow Expulsion

Notice that blocks 'a' and 'b' remain unchanged from their original positions.

## Record Chaining Using Relative Pointers

When a keyed file is less than 32 megabytes (or less than 65,535 blocks) in size, the blocks are chained together using absolute pointers. This means that the pointers are actual block numbers. However, once the keyed file exceeds 65,535 blocks, the pointers cannot point to a block beyond the 65,535th block. In order to allow for larger keyed files, relative pointers are used once the file size exceeds 65,535 blocks.

Relative pointers are relative offsets from the home block. Therefore, if the home block for a particular chain is 153, a relative pointer of 60 is pointing to block 213. A relative pointer of -50 is pointing to block 103. Additionally, in order to avoid pointers that are equal to 0, if a pointer is pointing to the home block in the chain, the pointer has a value of -32,768 instead of 0.

When adding a record to a file that uses relative pointers, if a new block needs to be added to the chain, the file is searched forward from the home block up to 32,767 blocks. If the search reaches the end of the file, the search is terminated. If no block is found in the forward search, the file is searched backwards from the home block up to 32,767 blocks. If the search reaches the beginning of the file (block 1), the search is terminated.

Below is an example of a chain of blocks in a keyed file that uses relative pointers. In this example, block 11 is the home block. The order of the blocks in this example chain is 11, 14, 44, 64, 5.

Table 18. Chain of Blocks in a Keyed File Using Relative Pointers (example)

Block 5	F=0 * *	B=53	Block 5 is the end of the chain, so the forward pointer is equal to 0. Block 5 points back to block 64 (home block + 53 = 64).
Block 11	F=3 * *	B=0	Block 11 is the home block. It points forward to block 14 (home block +3 = 14). The backward pointer is 0 since this is the home block.
Block 14	F=33 * *	B=-32,768	Block 14 points forward to block 44 (home block + 33 = 44). The block points backward to the home block, so the backward pointer is -32768.
Block 44	F=53 * *	B=3	Block 44 points forward to block 64 (home block + 53 = 64). The block points backward to block 14 (home block +3 = 14).
Block 64	F=-6 * *	B=33	Block 64 points forward to block 5 (home block - 6 = 5). The backward pointer points to block 44 (home block + 33 = 44).

## Keyed File Control Record

The Keyed File Control Block (KFCB) is located in sector zero of a keyed file. When creating a keyed file, initialize the entire sector (512 bytes) to zero and then initialize the appropriate fields.

All fields marked with an asterisk (\*) should be initialized (zero might be valid for some of these fields). All binary fields are in 80286 convention (for example, decimal 512 would be B'0002').

Do not use any of the reserved fields.

Offset	Length	Description
* 0	12	Reserved
12	18	File name (ASCII with binary zero padded)



The following table contains keyed file creation time stamp:

Offset	Length	Description
* 30	2	Year the file was created (binary)
* 32	1	Month the file was created (binary)
* 33	1	Day the file was created (binary)
* 34	4	Time the file was created in milliseconds after midnight (binary)
* 38	2	Time zone (can be left zero)

The following table contains keyed file parameters:

Offset	Length	Description
* 40	2	Block size (must be 512, B'0002')
* 42	4	Number of blocks in the file
* 46	2	Keyed record size (cannot exceed block size)
* 48	4	Randomizing divisor
* 52	2	Key offset (must be zero)
* 54	2	Key length (cannot exceed record size)
* 56	4	Chaining threshold

The following table contains keyed file statistics:

Offset	Length	Description
60	4	Longest chain encountered
64	4	Total failed requests (including key not found)
68	4	Total reads (not for update)
72	4	Total reads for update
76	4	Total writes to add records
80	4	Total writes to update records
84	4	Total release requests
88	4	Total delete requests
92	4	Total reads or writes of records chained 1 deep
96	4	Total reads or writes of records chained 2 deep
100	4	Total reads or writes of records chained 3 deep
104	4	Total reads or writes of records chained > 3 deep
108	4	Total file opens
112	4	Total file closes without release
116	4	Total file closes with release
120	10	Time stamp when statistics were cleared (format the same as the create time stamp above)
130	30	Reserved

The following table contains miscellaneous information:

Offset	Length	Description
* 160	9	File valid string (if valid = FSFACADX)
169	10	Reserved
* 179	1	Hashing algorithm (binary 0,1, or 2)
* 180	2	Chain pointer type used for the file (binary 0 or 1). If 0, chain pointers are absolute block numbers. If 1, chain pointers are relative offsets from the home block. The range is $\pm 32767$ . A chain pointer to the home block is -32768.
182	266	Reserved
448	60	For use by user applications
* 508	4	User identification (This should be "USER" or another identification of how the file was created. This field is binary zeros if created by Toshiba.)

---

## Chapter 7. Using the Input Sequence Table Build Utility

This chapter is your guide to using the Input Sequence Table Build Utility for building and maintaining your application's input sequence tables. The tables' concepts and definitions are described in Chapter 3, "Programming Terminal I/O Devices," on page 39 in the I/O Processor section. You should review that section if you are not familiar with input sequence tables. This utility has extensive help panels, that are a valuable resource for learning about the input sequence tables.

---

### Input Sequence Tables

The input sequence tables consist of three tables:

- Input state table
- Label format table
- Modulo check table

The input state table is always required to allow operator input from the keyboard, OCR, magnetic wand, or scanner. The input sequence table and modulo check table are optional depending on your application requirements.

The input sequence tables are used by the I/O processor to determine what operator input is allowed and how to process it.

---

### Using the Input State Table Utility

The Input Sequence Table Utility is a menu-driven program that enables you to add (build), erase, copy, change, activate, or rename any of these tables. You can also display, print, or file a formatted report for a table.

### Tables Maintained by the Utility

In addition to the three input sequence tables, the utility maintains a symbol table associated with each of the input sequence tables. The utility automatically names and updates the symbol tables as you name and update the input sequence table.

The utility also maintains a working table for each table you have changed but not activated. This function enables your application to use the input sequence table unchanged while you change the working tables. The utility automatically names and updates the working tables as you update the input sequence table.

When you use the utility options to manage your input sequence table, the utility also manages any associated symbol and working files. For this reason, you should always use the utility options to manage your tables rather than using other programs that are not aware of the additional tables.

---

### Running the Input Sequence Table Utility

To run the Input Sequence Table Utility, use the store controller system panels. Using these panels, you can add, display, erase, copy, print, change, or rename any of these tables. There is also an option that enables you to file (instead of printing) the report produced by a display or print option. To work on any other table, select the appropriate table on the INPUT SEQUENCE TABLE UTILITY panel.

---

### Input Sequence Table Utility on a LAN (MCF Network) System

On a LAN (MCF Network) system, you can run the Input Sequence Table Utility on any store controller. The utility creates only local files (nondistributed files). You must use the Distribution File Utility to distribute the results of the Input Sequence Table Utility.

When you use the Copy option from the INPUT SEQUENCE TABLE panel to copy a table, the new table is a local file regardless of the attributes of the source file.

If file security is enabled on your system, you cannot change an input sequence table in the ADX\_IPGM subdirectory. The Input Sequence Table Utility executes as user ID = 1 and group ID = 3, and can therefore only create files in the ADX\_UPGM subdirectory or in the root. While this is important to protecting your files, it can be inconvenient when updating a table in the ADX\_IPGM subdirectory. To overcome this temporary inconvenience, you can copy input sequence table files produced by the utility from the ADX\_IPGM subdirectory to the ADX\_UPGM subdirectory by using the Copy option on the INPUT SEQUENCE TABLE panel.

Use the following method to update a table in the ADX\_IPGM subdirectory:

1. Run the Input Sequence Table Utility as described in “Running the Input Sequence Table Utility” on page 197.
2. Copy the table from the ADX\_IPGM subdirectory to the ADX\_UPGM subdirectory using the input sequence table Copy option.
3. Make the necessary changes using the Change option.
4. Activate the changes using the Input Sequence Table Utility.
5. Use the ASM Utility to update the table to the ADX\_IPGM subdirectory. For information on this utility, see the *4690 OS: User's Guide*.

## Input Sequence Table Utility Options

The following options can be used for any of the three input sequence tables:

### Display a table

The display option displays a formatted report of a table. For the input state table you can request a report for part of the table.

### Print a table

The print option prints a formatted report of a table. For the input state table you can request a report for part of the table.

### File the report of a table

The file option writes a formatted report of a table to a file name you choose. For the input state table you can request a report for part of the table. The name for this file must be in the root or in ADX\_UPGM. You are responsible for erasing the report file.

### Erase a table

When you erase a table, the system erases the named table and any associated symbol and working tables. The system asks you for a confirmation of your option choice before erasing the table.

### Copy a table

This option copies a table and associated symbol and working tables.

### Rename a table

This option changes the name of a table and associated symbol and working tables. Before renaming a table, make sure that you do not execute any applications that refer to the table by its old name.

### Change a table

This option enables you to change an existing table and still use the unchanged table. When you change a table, the utility enables you to change the working table if one already exists. If there is no working table, one is created by making a copy of the table. You make changes to the working table while your applications continue to use the original one.

You can change a portion of a table, leave the utility, and return to work on it later. The utility maintains your changes in the working table. The utility does not make the changed table active until you request the option to activate a table.

### Activate a table

This option enables you to replace the currently active table with the associated working table. When you activate the working table, the utility erases the currently active table and associated symbol table and renames the working table and working symbol table to the names of the original tables. The utility cannot erase a file in ADX\_IPGM due to file security.

### Add a new table

This option enables you to build a new table. Because of references between tables, the input sequence table should be created in the following order:

1. Label format table
2. Modulo check table
3. Input state table

## Table Naming Conventions

The table names are also the file names of the tables. When naming your tables, use the following naming conventions. Also see “Naming files and subdirectories” on page 16. Use a file name that places files in the root or in ADX\_UPGM.

*xxxa@bbb.DAT*

where:

- xxx* = Three characters of your choice. If you are defining additional tables for the Toshiba licensed products, use UUU.
- a* = A character of your choice.
- @* = The fifth character of each file name.
- bbb* = Three characters of your choice. These characters are optional.

The utility automatically names symbol and working tables. The symbol table name is created by replacing the @ character with the \$ character. The working table names are created by replacing the @ character with a character and replacing the \$ character with a character.

## Notes on Using the Utility

When you run the Input Sequence Tables Utility, it should be the only utility or application being executed due to the utility storage requirements.

When you are adding states or function codes and you name a next state, the named state must have already been defined or you receive an error message. To bypass the error, you can temporarily call the next state CURRENT or LOCK. You can then change to the correct state name after the state has been added.

You should assign state IDs sequentially starting with 1. Allowing gaps in ID assignments causes additional storage to be used.

Once a state ID is assigned, you cannot change the ID. You can change only the state symbolic name.

When you are entering messages to be displayed, there is a difference between spacing with the space bar and moving the cursor. When you use the space bar, a blank is displayed in that position. When you move the cursor, nothing is displayed for that position (unless you previously entered something).

When you choose to add to a table, default values appear in the panel input fields. When you add a state, you can request another state be used as a model. This model state is used for default values. When you

add function codes to a state or common state information, the previous function code you added for the state or common state is used to set up default values.

The maximum size of any table is 64 KB. You can use the directory function of the operating system to determine the size of your tables.

---

## Chapter 8. Using the LIB86 Library Utility

This chapter tells you how to use the LIB86 Library Utility. It includes information on creating library files, appending an existing library, and replacing or deleting library modules.

A library file consists of one or more object modules. To increase the speed of the linking process, a library file contains an index. The index contains all the public functions and subprograms that are in each module, enabling LINK86 to determine which routines in the library are required to create the program. LIB86 is a library utility that enables you to create and modify your own library files.

The following files are on the 4690 Optionals:

- LIB86.286 is the operating system library utility.
- LIB86.EXE is the DOS or OS/2 library utility.

LIB86 is a versatile library manager for developing library files to use with LINK86. LIB86 can:

- Create a library file from a group of object files
- Append modules to an existing library file
- Replace modules in an existing library file
- Delete modules from an existing library file
- Select specific modules from a library file
- Display library information

LIB86 generates library files according to instructions you specify in the command line. Input files can have a file extension of OBJ or L86. LIB86 assumes an input file has an OBJ file extension if you do not specify a different file extension. To conclude processing, LIB86 displays the total number of modules processed and the use factor. The use factor is a decimal percentage value specifying the amount of memory LIB86 uses.

The command line starts LIB86 and specifies the input files to be processed.

A LIB86 command line uses the following general format:

```
A:>LIB86 filespec=filespec {[options]} {,filespec}
```

LIB86 checks for errors and displays literal error messages. The error messages are described in “LIB86 Error Messages” on page 731.

---

### Using LIB86 Command-Line Options

LIB86 has 14 command-line options. You specify the options (enclosed in brackets) in a LIB86 command line. Table 19 on page 201 lists the LIB86 command-line options, an option abbreviation, and a brief description of each option.

*Table 19. LIB86 Command-Line Options*

Option	Abbr	Purpose
DELETE	D	Delete a module from a library file.
Echo	D	Echos contents of the INP file on the console.
EXTERNALS	E	Show external symbols in a library file.
INPUT	I	Read commands from input file.
MAP	MA	Create a module map.
MODULES	MO	Show modules in a library file.
NOALPHA	N	Show modules in order of occurrence.

Table 19. LIB86 Command-Line Options (continued)

Option	Abbr	Purpose
PUBLICS	P	Show public symbols in a library file.
REPLACE	R	Replace a module in a library file.
SEGMENTS	SEG	Show segments in a module.
SELECT	SEL	Select a module from a library file.
XREF	X	Create a cross-reference file.
\$O		Specify input file location.
\$X		Specify .XRF file destination.
\$M		Specify .MAP file destination.

You can specify either the option or its abbreviation in a command line. The remainder of this chapter explains the use of command-line options.

LIB86 can process any number of files. The length of a command line can be a maximum of 128 characters. When you must use a command line that exceeds 128 characters, you can shorten file names, or place the command line in a file and use the INPUT option.

You can create command-line files with an INP file extension using a text editor. Enter the new library name before an equal sign and list each input file, with options, after the equal sign, the same way you do in a command line. Do not include the characters LIB86 in the file. You can place tab characters, carriage returns, and line feeds anywhere in a command line file.

The following example shows the beginning of a command-line file named LIBRARY1.INP:

```
LIBRARY1 = SUBTOT [XREF], ADD2, SUB45, MULT, DIV2,
          NET1, NET2, NET3,
          TOTAL, GROSS1, GROSS2, GROSS3,
          CHART1, CHART2, CHART3,
          .
          .
          .
```

To use the command-line file and start LIB86, specify the INP file as follows:

```
A:>LIB86 LIBRARY1 [INPUT]
```

**Note:** LIB86 assumes an INP file extension if you are using the INPUT option.

The preceding example specifies the INPUT option, instructing LIB86 to read the remainder of the command line from the LIBRARY1.INP disk file.

---

## Creating a Library File

The following example creates a library named TEST.L86 from the input files ONE.OBJ, TWO.OBJ, and THREE.OBJ. You do not have to specify the L86 file extension for the library name. LIB86 assumes a file extension of OBJ for input files unless you specify a different file extension. OBJ files contain only one module.

```
A:>LIB86 TEST=ONE,TWO,THREE
```

You can create one large library file from several smaller library files.

The following example creates a large library file named TESTLIB.L86 from the input files LIB1.L86 and LIB2.L86. L86 files contain more than one module.



```
A:>LIB86 TESTLIB=LIB1.L86,LIB2.L86
```

You can combine OBJ and L86 files in one command line to create a library, as shown in the following example:

```
A:>LIB86 MATHLIB=MULT,DIVIDE.L86
```

The preceding example creates a library file named MATHLIB.L86 from the input files MULT.OBJ and DIVIDE.L86.

---

## Appending an Existing Library

To add a module to an existing library, specify the existing library file name on both sides of the equal sign. Then, list the input files you want to append. Include the L86 file extension for the library file name on the right side of the equal sign.

The following example appends the files ONE.OBJ and LIB1.L86 to the existing library file TESTLIB.L86:

```
A:>LIB86 TESTLIB=TESTLIB.L86,ONE,LIB1.L86
```

You can rename an appended library file, as shown in the following example:

```
A:>LIB86 NEWTEST=TESTLIB.L86,ONE,LIB1.L86
```

The preceding example appends the files ONE.OBJ and LIB1.L86 to the existing library TESTLIB.L86, creating a new library file named NEWTEST.L86.

---

## Replacing Library Modules

Use the REPLACE option to replace a module in an existing library file.

The following command line replaces the module ONE with the file NEWONE.OBJ in the library file TESTLIB.L86.

**Note:** See the correct use of brackets in the command line.

```
A:>LIB86 TESTLIB=TESTLIB.L86 [REPLACE [ONE=NEWONE]]
```

If you want to replace a module but maintain the same module name, specify the name only once after the REPLACE option.

The following example replaces the module ONE with a new ONE.OBJ file in the library TESTLIB.L86, and renames the library NEWLIB.L86:

```
A:>LIB86 NEWLIB=TESTLIB.L86 [REPLACE [ONE]]
```

You can replace several modules with one command line. Separate the REPLACE option specifications with commas, as shown in the following example:

```
A:>LIB86 NEWLIB=TESTLIB.L86 [REPLACE [ONE=NEW1, TWO=NEW2]]
```

You cannot use the command-line options DELETE and SELECT with REPLACE in the same command line.

LIB86 displays an error message if it cannot find a specified module in the library file.

---

## Deleting Library Modules

Use the DELETE option to delete modules from an existing library file as shown in the following example. Module TWO is deleted from the library file TESTLIB.L86:

```
A:>LIB86 TESTLIB=TESTLIB.L86 [DELETE [TWO]]
```

You can delete several modules with one command line. Separate modules after the option DELETE with commas.

The following example deletes three modules to create a new library named NEWLIB.L86:

```
A:>LIB86 NEWLIB=TESTLIB.L86 [DELETE [ONE, TWO, FIVE]]
```

You can delete a group of contiguous library modules using a hyphen, as shown in the following example:

```
A:>LIB86 NEWLIB=TESTLIB.L86 [DELETE [ONE - FIVE]]
```

The preceding command line deletes all modules from module ONE through module FIVE.

You cannot use the command-line options REPLACE and SELECT with the DELETE option in one command line.

LIB86 displays an error message if it cannot find a specified module in a library file.

---

## Selecting Modules

Use the SELECT option to select specific modules from an existing library to create a new library.

The following example creates a new library named NEWLIB.L86 that consists of three modules selected from OLDLIB.L86:

```
A:>LIB86 NEWLIB=OLDLIB.L86 [SELECT [TWO, FOUR, FIVE]]
```

You can select a group of contiguous library modules using a hyphen, as shown in the following example. A new library is created that consists of five modules selected from an existing library, assuming modules ONE, TWO, THREE, FOUR, and FIVE are contiguous in the library file.

```
A:>LIB86 NEWLIB=OLDLIB.L86 [SELECT [ONE - FIVE]]
```

You cannot use the command-line options DELETE and REPLACE with the SELECT option in the same command line.

LIB86 displays an error message if it cannot find a specified module in a library file.

---

## Displaying Library Information

LIB86 can produce listing files of two types: a cross-reference file and a library module map. A cross-reference file contains an alphabetized list of all public, external, and segment name symbols a library file. Following each symbol is a list of all modules that contain the symbol. LIB86 marks the module or modules that define the symbol with a pound (#) sign. LIB86 encloses segment names in slashes (/). For example, the segment CODE would appear as /CODE/.

Use the XREF option to create a cross-reference listing for a specified library file.

The following example creates a cross-reference file named TESTLIB.XRF for the TESTLIB.L86 library file:

```
A:>LIB86 TESTLIB.L86 [XREF]
```

A module map contains an alphabetized list of all modules in a library file. Following each module name is a list of all segments in the module and the length of each segment. A module map also includes a list of all public and external symbols specified in the module.

Use the MAP option to create a module map for a specified library file.

The following example creates a module map named TESTLIB.MAP for the TESTLIB.L86 library file:

```
A:>LIB86 TESTLIB.L86 [MAP]
```

Usually, LIB86 alphabetizes the names of modules in a module map listing. Use the NOALPHA option to produce a module map that lists module names in order of occurrence, as shown in the following example:

```
A:>LIB86 TESTLIB.L86 [MAP, NOALPHA]
```

You can use LIB86 to create partial library information maps using the MODULES, SEGMENTS, PUBLICS, and EXTERNALS options. You can use the four options in any combination.

The following example creates a module map that contains only public and external symbols:

```
A:>LIB86 TESTLIB.L86 [PUBLICS, EXTERNALS]
```

You can combine the SELECT option with any of the options previously described to generate partial library information maps, as shown in the following examples:

```
A:>LIB86 TESTLIB.L86 [XREF, [SELECT [ONE, + TWO, THREE]]]  
A:>LIB86 MATHLIB.L86 [MAP, NOALPHA, SELECT [MULT, + DIVIDE]]  
A:>LIB86 LIBRARY1.L86 [MODULES, SEGMENTS, SELECT + [ONE - FIVE]]
```

---

## Accessing Files in Other Directories

LIB86 assumes that all files specified on a command line (or INP file) are in the default directory. You can access files in other directories in several ways. These options are listed in their order of precedence:

1. A fully specified path name can be included with each L86 or OBJ file name. The following example shows how to specify locations of L86 or OBJ files that are not contained in the default directory:

```
A:>LIB86 C:\LIBA\NEW1=LIB1.L86,C:\OBJ1\ONE,C:\OBJ2\TWO
```

In this case, a new library, NEW1.L86, will be created in the C:\LIBA directory. The components of the new library will be LIB1.L86 (default directory), ONE.OBJ (C:\OBJ1 directory), and TWO.OBJ (C:\OBJ2 directory).

2. Options \$M, \$O, and \$X can be used on the command line (or in an INP file). These options override the default directory. They must be specified as follows:
  - \$Ofilespec specifies input OBJ or L86 file location.
  - \$Xfilespec specifies output XRF file destination.
  - \$Mfilespec specifies output MAP file destination.

The \$O option remains in effect as the library utility processes a command line from left to right, until it encounters another \$O. This feature is useful when you create a library from groups of files in different directories.

The following command causes a MAP file to be placed in C:\MAPS, an XRF file to be placed in C:\XREFS, and looks for OBJ files only in the C:\OBJS subdirectory.

```
A:>LIB86 TEST.L86[X,MA,$XC:\MAPS,$MC:\XREFS,$OC:\OBJS]
```

**Note:** The \$O option is selectively ignored if a fully specified file name is used for any OBJ, INP, or L86 input file.

3. LIB86 recognizes 4690 logical file names, but does not supply the .OBJ file extension when an extension is not specified.
4. A search path can be set up to look for OBJ, L86, or INP files if they are not found in the default directory. Environment variable, LIB86PATH, should be set (or defined) in the current DOS, OS/2, or 4680 session before running the LIB86 utility.

If you want the library utility to search for OBJ, INP, or L86 components first in the C:\NEWCODE, then in the C:\OLDCODE directories, establish that search path by issuing the following command:

OS/2 or DOS:

```
A:>SET LIB86PATH=C:\NEWCODE;C:\OLDCODE
```

4680:

```
A:>DEFINE LIB86PATH=C:\NEWCODE;C:\OLDCODE
```

When the library utility is subsequently run, OBJ, INP, and L86 files will be searched for along this path if they are not found in the default directory.

**Note:** This option is selectively overridden if either option 1 or option 2 is specified for specific input files.

---

## Chapter 9. Using the Linker Utility and the POSTLINK Utility

---

### The Linker Utility

The Linker Utility, LINK86, is a linkage editor that combines relocatable object files into a load module that runs on the operating system. Any 4690-compatible compiler or assembler can produce the object files. See Table 21 on page 210 for a summary of command-file option parameters.

The linker is available to run in a DOS, OS/2, or the operating system environment. The file LINK86.286 runs in the operating system environment. LINK86.EXE runs in either a DOS (Version 3.3 or later) or OS/2 (Version 1.2 or later) environment. The linker utility is shipped in the 4690 Optionals.

LINK86 accepts the following types of files as input:

#### **Object (OBJ) file**

A language source file that has been translated by the compiler or assembler into a machine-readable object code.

#### **Library (L86) file**

An indexed library of commonly used object modules. The library utility LIB86 generates library files.

#### **Input (INP) file**

A file that contains file names and options, the same as a command line you enter from the keyboard.

LINK86 creates the following types of files:

#### **Executable Load Module (286) file**

Contains a memory image of a program and runs directly under the operating system.

#### **Overlay (OVR) file**

Contains information that is loaded into memory when it is needed by the program.

#### **Symbol Table (SYM) file**

Contains a list of symbols from the object files and their offsets.

#### **Line Number (LIN) file**

Contains a list of code offsets of program source lines. This file is created only when the compiler puts line number information into the object files being linked.

#### **Map (MAP) file**

Contains segment information about the load module.

#### **Debug Information (DBG) file**

Contains information used by the 4690 application debugger.

During processing, LINK86 displays unresolved symbols. An unresolved symbol is declared to be external in one or more modules, but is not publicly defined in any module.

When processing is complete, LINK86 displays the size of each section of the load module. See Figure 13 on page 208.

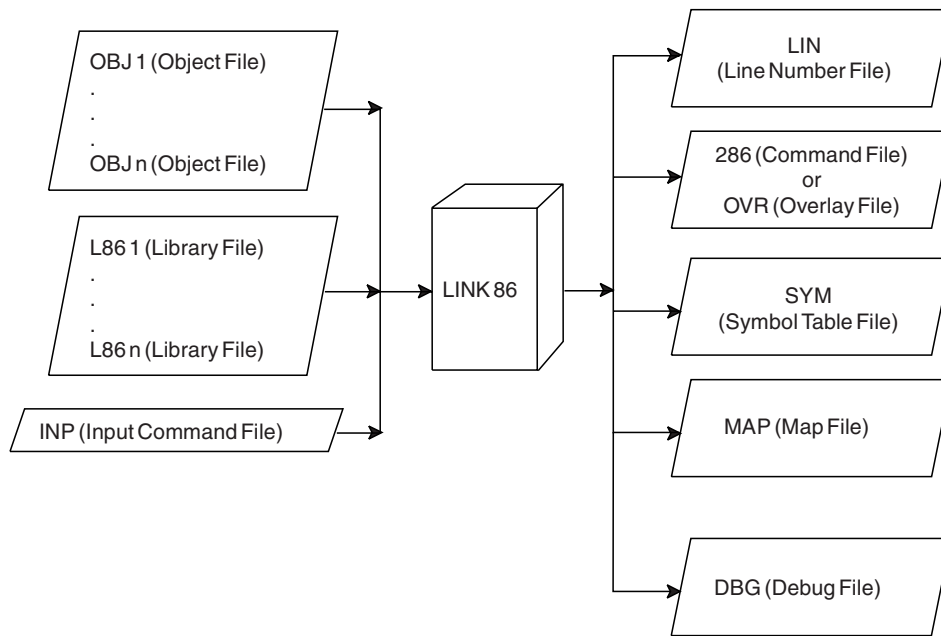


Figure 13. LINK86 Operation

## LINK86 Command Syntax

You invoke LINK86 with the following command format:

```
LINK86 [filespec =] filespec1 [,filespec2,...,filespecn]
```

*filespec* is a file specification, consisting of an optional path specification and a file name with optional file extension. If you enter a file name to the left of the equal sign, LINK86 creates output files with that file name and the appropriate file extensions.

For example, using the files PARTA, PARTB, and PARTC, the following command creates MYFILE.286:

```
A:>LINK86 MYFILE = PARTA,PARTB,PARTC
```

**Note:** The files PARTA, PARTB, and PARTC can be a combination of object files and library files. If no file extension is specified, the linker assumes a file extension of OBJ.

If you do not specify an output file name, LINK86 creates the output files using the first file name in the command line.

For example, the following command creates the file PARTA.286:

```
A:>LINK86 PARTA,PARTB,PARTC
```

**Note:** If you specify a library file in your link command, do not enter the library file as the first file in the command line.

You can also instruct LINK86 to read its command line from a file, thus making it possible to store long or commonly used link commands on disk:

```
A:>LINK86 LINKFIL[I]
```

Additional linker options are discussed later in this chapter.

## Linking With Shared Run-time Libraries

LINK86 supports shared run-time libraries. Shared run-time libraries (SRTLs) enable multiple users to share a single copy of library code at run time. It is not necessary for each user to store library code in their load module. When libraries are shared, only references to the library code are linked with a user's object files.

No extra steps are required when linking BASIC object files with SRTL files. Do not specify the runtime libraries SB286L.L86 or SB286TVL.L86. They are automatically requested by the compiler.

**Note:** See “SHARED and NOSHARED Options” on page 213 for information on linking with shared run-time libraries.

## LINK86 Command Options

When you invoke LINK86, you can specify command options that control the link operation. Command options are in one of three categories, depending on the type of file affected.

Command options in the first category affect the contents of the command file and apply to the entire link operation. Command options in the second category affect the SYM and MAP files. These options turn on and off as LINK86 processes the command line from left to right. Command options in the third category affect the library and input files and apply to one file in the command line.

When you specify command options, enclose them in square brackets following the file name.

A command option is specified using the following command format:

```
A:>LINK86 file[option]
```

For example, to specify the command option MAP for the file TEST1 and the NOLOCALS option for the file TEST2, enter the following:

```
A:>LINK86 TEST1[MAP],TEST2[NOLOCALS]
```

You can use spaces to improve the readability of the command line, and you can put more than one option in square brackets by separating them with commas.

The following example specifies that the MAP and NOLOCALS options be used for the TEST1 file and the option LOCALS for the TEST2 file:

```
A:>LINK86 TEST1 [MAP, NOLOCALS], TEST2 [LOCALS]
```

Table 20. LINK86 Command Options and abbreviations

Option	Abbreviation	Meaning
CODE	C	Controls contents of CODE section of load module.
DATA	D	Controls contents of DATA section of load module.
STACK	ST	Controls contents of STACK section of load module.
LIBSYMS	LI	Include symbols from library files in SYM file.
NOLIBSYMS **	NOLI	Do not include symbols from library files in SYM file.
LOCALS **	LO	Include local symbols in SYM file.
NOLOCALS	NOLO	Do not include local symbols in SYM file.
NOSYMS	NOSY	Inhibits generation of a SYM file.
LINES **	LIN	Create LIN file with line number information.
NOLINES	NOLIN	Do not create LIN file.
NOSEARCH	NOSE	Link all library modules whether referenced or not.

Table 20. LINK86 Command Options and abbreviations (continued)

Option	Abbreviation	Meaning
MAP	M	Create a MAP file.
SEARCH **	S	Search library and only link referenced modules.
SHARED	SH	Force an SRTL to be treated as shared.
NOSHARED	NOSH	Force an SRTL to be treated as a normal unshared library.
CODESHARED	CODESH	Designates that this load module's code can be shared by multiple processes. After a load module is linked CODESHARED, the load module must be POSTLINKed using the POSTLINK Utility.
INPUT	I	Read command line from disk file.
ECHO	ECHO	Echo contents of INP file to the screen.
DBI	DBI	Create a DBG file that contains debug information.

**Note:** \*\* = Default Value

## Command-File Options

The options described in this section affect the contents of the command file created by LINK86. Table 21 on page 210 lists the command-file option parameters.

Table 21. Command-File Option Parameters

Parameter	Abbreviation	Meaning
GROUP	G	Groups to be included in load module section
SEGMENT	S	Segments to be included in load module section
ADDITIONAL	AD	Additional memory allocation for the load module section
MAXIMUM	M	Maximum memory allocation for load module section

**CODE/DATA/STACK:** A load module consists of a 128-byte header record ordinarily followed by three sections. The sections are called CODE, DATA, and STACK. Each of the sections correspond to a LINK86 command option of the same name. The header contains the length of each section of the load module and its minimum and maximum memory requirements. The operating system uses the header to load the file.

## GROUP and SEGMENT

The GROUP and SEGMENT parameters each contain a list of groups or segments that you want LINK86 to put into a specified section of the load module.

For example, the following command instructs LINK86 to put the segments CODE1, CODE2, and all the segments in group XYZ into the CODE section of the file TEST.286:

```
A:>LINK86 TEST [CODE [SEGMENT [CODE1, CODE2], GROUP [XYZ]]]
```

## ADDITIONAL and MAXIMUM

The ADDITIONAL and MAXIMUM parameters tell LINK86 the values to put in the load module header. These parameters override the default values that LINK86 uses.

Each parameter is a hexadecimal number enclosed in square brackets.



The **ADDITIONAL** parameter specifies the amount of additional memory, in paragraphs, required by the specified section of the load module.

**Note:** A paragraph is 16 (X'10') bytes.

The **MAXIMUM** parameter specifies the maximum amount of memory needed by the section of the load module. The program loader attempts to allocate as much of the requested maximum as possible.

In the following example, the command creates the file TEST.286:

```
A:>LINK86 TEST [DATA [ADD [100], MAX[FFF]]]
```

The TEST.286 file header contains the following information:

- The DATA section requires at least X'100' paragraphs in addition to the data in the load module.
- The DATA section can use up to X'FFF' paragraphs of memory.

## **SYM File Options**

The following command options affect the contents of the SYM file that LINK86 creates:

- NOSYMS
- LOCALS
- NOLOCALS
- LIBSYMS
- NOLIBSYMS

These options must appear in the command line after the specific file or files to which they apply. When you specify one of these options, it remains in effect until you specify another option. Therefore, if a command line or input file (INP) contains two options, the leftmost option affects all of the listed files until the next option is encountered. The next option affects all remaining files specified on the command line or input file.

**NOSYMS:** The NOSYMS option prevents the generation of a SYM file. In this case, all other SYM file options are ignored.

**LOCALS and NOLOCALS:** The LOCALS option directs LINK86 to include local symbols in the SYM file if they are in the object files being linked. The NOLOCALS option instructs LINK86 to ignore local symbols in the object files. The default is LOCALS.

For example, the following command creates a SYM file containing local symbols from TEST2.OBJ and TEST3.OBJ, but not from TEST1.OBJ:

```
A:>LINK86 TEST1 [NOLOCALS], TEST2 [LOCALS], TEST3
```

**LIBSYMS and NOLIBSYMS:** The LIBSYMS option instructs LINK86 to include in the SYM file symbols from a library searched during the link operation. The NOLIBSYMS option instructs LINK86 not to include library symbols in the SYM file. A library search usually involves the runtime subroutine library of a high-level language such as 4680 BASIC. Because the symbols in such a library are usually of no interest to the programmer, the default is NOLIBSYMS.

The following example disables library symbol generation for ADXADMBL.L86:

```
A:>LINK86 TEST,ADXACRCL.L86,ADXADMBL.L86 [NOLI]
```

## **LIN File Options**

The 4680 BASIC compiler provides an option that puts line numbers into object files. If line numbers are present in the object file, LINK86 can create a file containing line numbers and their code offsets.

The **LINES** and **NOLINES** options specify whether or not LINK86 creates a LIN file.

The LINES option, which is active by default, instructs LINK86 to create a LIN file, if possible. If no line information is present in the object file, LINK86 does not create the LIN file. The NOLINES option instructs LINK86 not to create a LIN file, even if line numbers are present in the object file:

```
A:>LINK86 TEST [NOLIN]
```

## MAP File Options

The MAP option instructs LINK86 to create a MAP file containing information about segments in the load module.

The amount of information that LINK86 puts into the MAP file is controlled by the following optional parameters:

OBJMAP	NOOBJMAP
L86MAP	NOL86MAP
ALL	NOCOMMON

The optional parameters are enclosed in brackets following the MAP option. The OBJMAP parameter directs LINK86 to put segment information about OBJ files in the MAP file. The NOOBJMAP parameter suppresses this information. The L86MAP parameter instructs LINK86 to put segment information from L86 files into the MAP file. The NOL86MAP parameter suppresses this information. The ALL parameter instructs LINK86 to put all the information into the MAP file. The NOCOMMON parameter suppresses all common segments from the MAP file.

When you instruct LINK86 to create a MAP file, you can change the parameters to the MAP option at different points in the command line. For example, the following command directs LINK86 to create a map file containing segment information from FINANCE.OBJ and SCREEN.L86:

```
A:>LINK86 FINANCE [MAP[ALL]],SCREEN.L86,GRAPH.L86 [MAP[NOL86MAP]]
```

### Notes:

1. Segment information for GRAPH.L86 is suppressed by the NOL86MAP option.
2. If you specify the MAP option with no parameters, LINK86 uses OBJMAP and NOL86MAP as defaults.

## L86 File Options

The following command options determine how the library files are used by LINK86:

- SEARCH
- NOSEARCH
- SHARED
- NOSHARED

**SEARCH and NOSEARCH Options:** The SEARCH option instructs LINK86 to search the preceding library, and include in the load module only those modules that satisfy external references from other modules. Because SEARCH is the default value, using it as a value is redundant. The NOSEARCH option instructs the linker to include in the load module all modules whether an external reference is satisfied or not.

**Note:** LINK86 searches L86 files automatically.

The NOSHARE option must be specified for each module to be linked.

For example, the following command creates the file TEST1.286 by combining the object files TEST1.OBJ, TEST2.OBJ, and modules from MATH.L86 that are referenced in TEST1.OBJ or TEST2.OBJ:

```
A:>LINK86 TEST1, TEST2, MATH.L86
```

The modules in the library file do not have to be in any special order. LINK86 makes multiple passes through the library index when attempting to resolve references from other modules.

**SHARED and NOSHARED Options:** The SHARED and NOSHARED options determine whether a library file is to be used as an SRTL. When a runtime library is NOSHARED, both the code and the data from that library are linked with the object files. When a runtime library is SHARED, a single copy of the library code resides in a special load module called an executable shared runtime library (XSRTL). The code stored in an XSRTL file can be accessed by any file linked as a user of the SRTL.

When an SRTL is created, it is given an attribute that determines whether the library is to be treated as a SHARED or a NOSHARED option.

The store controller runtime library for 4680 BASIC (SB286L.L86) was created with the SHARED attribute.

If an SRTL has a default attribute of SHARED, you can force LINK86 to treat it as a normal library by specifying the NOSHARED option. This forces the referenced SRTL routines to be resident in the user's code file, and the loader does not have to perform a load-time resolution of external references.

As an example of the NOSHARED option, the following command format causes LINK86 to treat the shareable runtime library as an unshared library:

```
A:>LINK86 MYPROG=MAIN,PART1,PART2[NOSHARED]
```

**CODESHARED:** Normally a load module created by the linker has a bit set in the header that identifies to the loader that only one process can use the code segments at one time. By specifying the CODESHARED option, you can force the loader to use the same copy of the module's code segments for multiple processes.

**Note:** A load module that has been linked with the CODESHARED option must be POSTLINKed using the POSTLINK Utility before the operating system will allow it to run.

If you are executing the same application program in both the Model 1 and Model 2 terminals, you should use this option because it saves a significant amount of storage. If your application uses the Display Manager product, you must *NOT* use this option because the Display Manager's code is not shareable.

## INPUT File Options

The INPUT and ECHO command options determine how LINK86 uses the input file.

The INPUT option instructs LINK86 to obtain further command-line input from the file. Other files can appear in the command line, but the input file must be the last file name on the command line. When LINK86 encounters the INPUT option, it stops scanning the command line entered from the keyboard.

**Note:** You cannot nest command input files. A command input file cannot contain the INPUT option.

The input file consists of file names and options the same as a command line entered from the keyboard. An input file can contain up to 4096 characters, including spaces but excluding comments. Comment delimiters recognized by LINK86 are ; and \*. When LINK86 encounters either of these delimiters in an input file, the remaining characters on that line are ignored. Use comments as often as you like to make the input file easier to understand and maintain.

In the following example, the file TEST.INP might include the lines:

```
;This is a comment
MEMTEST=TEST1,TEST2,TEST3,
IOLIB.L86[S],MATH.L86[S], **TThis is another comment
TEST4,TEST5[LOCALS]
```

To direct LINK86 to use this file for input, enter the following command format:

```
A:>LINK86 TEST.INP [INPUT]
```

If no file extension is specified for an input file, LINK86 assumes INP.

The ECHO option causes LINK86 to display the contents of the INP file on the display:

```
A:>LINK86 TEST [ECHO,I]
```

## Input/Output Option

The \$ option controls the source and destination devices under LINK86. The general form of the \$ option is:

*\$tpathname*

**Note:** *t* is a file type, and *pathname* is a fully specified path or a drive letter followed by a colon.

**File Types:** LINK86 recognizes the following seven file types:

- C - Load Module (286 or OVR)
- D - Debug Information File (DBG)
- L - Library File (L86)
- M - Map File (MAP)
- N - Line Number File (LIN)
- O - Object File (OBJ or L86)
- S - Symbol File (SYM)

The value of a \$ option remains in effect until LINK86 encounters a countermanding option as it processes the command line from left to right. For example, the following command will link TEST1.OBJ and TEST2.OBJ files from subdirectory C:\OBJ1 with the TEST3.OBJ file in subdirectory C:\OBJ2:

```
C:>LINK86 TEST.286=TEST1 [$OC:\OBJ1],TEST2,TEST3[ $OC:\OBJ2]
```

**\$C (Command) Option:** The \$C option uses the following format:

*\$Cpathname*

LINK86 usually creates the load module in the same subdirectory as the first object file in the command line. The \$C option instructs LINK86 to place the load module in the specified directory. The \$C option also applies to OVR files when you use LINK86 to create overlays.

```
A:>LINK86 TEST [$CC:\286S]
```

**\$D (Debug Information) Option:** The \$D option uses the following format:

*\$Dpathname*

LINK86 normally creates the DBG file in the default directory. The \$D option instructs LINK86 to place the DBG file in the directory you specified with the pathname.

```
A:>LINK86 TEST [$D:\DBGS]
```

**\$L (Library) Option:** The \$L option uses the following format:

*\$Lpathname*

LINK86 searches the default directory for runtime subroutine libraries that are linked automatically. The \$L option instructs LINK86 to search the specified *pathname* for the runtime subroutine libraries.

```
A:>LINK86 TEST [$LC:\LIBS]
```

**\$M (Map) Option:** The \$M option uses the following format:

*\$Mpathname*

LINK86 normally creates the MAP file in the default directory. The \$M option instructs LINK86 to place the MAP file in the specified directory.

```
A:>LINK86 TEST [$MC:\MAPS]
```

**\$N (Line Number) Option:** The \$N option uses the following format:

*\$Npathname*

LINK86 normally creates the LIN file in the same subdirectory as the load module. The \$L option directs the linker to place the LIN file in the directory specified by the *pathname* that follows the \$N option:

```
A:>LINK86 TEST [$NC:\LINS]
```

**\$O (Object) Option:** The \$O option uses the following format:

*\$Opathname*

LINK86 searches for the OBJ or L86 files that you specify in the command line on the default drive, unless such files have drive prefixes. The \$O option enables you to specify the drive location of multiple OBJ or L86 files without adding a path name prefix to each file name.

In the following example, the command instructs LINK86 that all the object files except the last one are located in subdirectory D:\OBJJS.

```
A:>LINK86 P [$OD:\OBJJS],Q,R,S,T,U.L86,B:V
```

**Note:** This does not apply to libraries that are linked automatically. See the \$L option.

**\$S (Symbol) Option:** The \$S option uses the following format:

*\$Spathname*

LINK86 normally creates the symbol file in the same subdirectory as the load module. The \$S option directs LINK86 to place this file in the subdirectory specified by the *pathname* that follows the \$S option.

```
A:>LINK86 TEST [$SC:\SYMS]
```

**DBINFO Option:** The DBINFO option instructs LINK86 to create a DBG file containing necessary information for the 4680/4690 Application Debugger. If you are combining several OBJ and/or L86 modules, specify the DBI option on the first OBJ file listed on the command line or in the INP file.

See the *4680-4690 Application Debugger User's Guide* for additional information about compiling and linking an application to prepare it for debugging.

```
A:>LINK86 TEST #DBI'
```

---

## Use of Link Path Variables to Search Other Directories

Sometimes you might want to search a particular path to find OBJ, L86, or INP files. If a file is not found in the first directory, the search continues to other directories in the specified path. Using this method, you can make modifications to a base set of OBJ files and keep the modified OBJ files in a separate directory or series of directories.

Using the LNK86PATH environment variable, you can define the search order so the linker looks first in the changed files directory and, if not found, then looks in the base directory.

The LNK86PATH environment variable is defined using the SET command (DOS or OS/2) or the DEFINE command when using 4690 OS V2 or higher. This variable must be set at least once before running the linker utility, and it must be set during the same session.

Under DOS or OS/2, use the following command:

```
A:>SET LNK86PATH=path1;path2;path3;<....;pathn>
```

where *pathn* represents the directories that you intend to search for files to be read by the linker.

Under the operating system, use the following command:

```
A:>DEFINE LNK86PATH=path1;path2;path3 <.... pathn>
```

where *pathn* represents the directories that you intend to search for files to be read by the linker.

---

## How Various Search Priorities Relate

The following search order is used when linking with files in remote directories:

1. A path name specified as part of a file name on the command line or in an INP file. If the file is not found, the search is abandoned.
2. The \$ directives \$O and \$L. If an OBJ, INP, or L86 file is not found in the specified directory, the search is abandoned.
3. The default directory.
4. The path specified by the environment variable LNK86PATH.

---

## Use of ERRORLEVEL Test

When running the LINK86 program from a batch file, the results of the link can be tested for errors. If the link step is successful, an ERRORLEVEL value of zero is returned. To test for an error, use the batch file statement "IF ERRORLEVEL 1...".

The following is an example within a batch file:

```
LINK86 TEST
IF ERRORLEVEL 1 ECHO We have a problem >> RESULTS
```

In the above example, if the link of the TEST file fails the message "We have a problem" is written to the file RESULTS.

---

## Overlays

Overlays are supported only in the store controller. This section describes how LINK86 creates programs with separate files called overlays. Each overlay file is a separate program module that is loaded into memory when it is needed by the program. By loading only those program modules that are needed at a particular time, the amount of memory used by the program is kept to a minimum; however, you must link your application with the runtime library using the NOSHARED option.

As an example, many application programs are menu-driven, with the user selecting the functions to perform. Because the program's modules are separate and invoked sequentially, they need not reside in memory simultaneously. Using overlays, each function on the menu can be a separate subprogram that is stored on disk, and loaded only when required. When one function is completed, control returns to the menu portion of the program. You then select the next function.

Figure 14 on page 217 shows the concept of using large program overlays. Assume that a menu-driven application program consists of three separate user-selectable functions. If each function requires 30 KB of memory, and the menu portion requires 10 KB, then the total memory required for the program is 100 KB (without overlays). However, if the three functions are designed as overlays (separate overlays), the program requires only 40 KB, because all three functions share the same locations in memory.

**Note:** The POSTLINK Utility must not be used on overlay files or the root module of an overlay file.

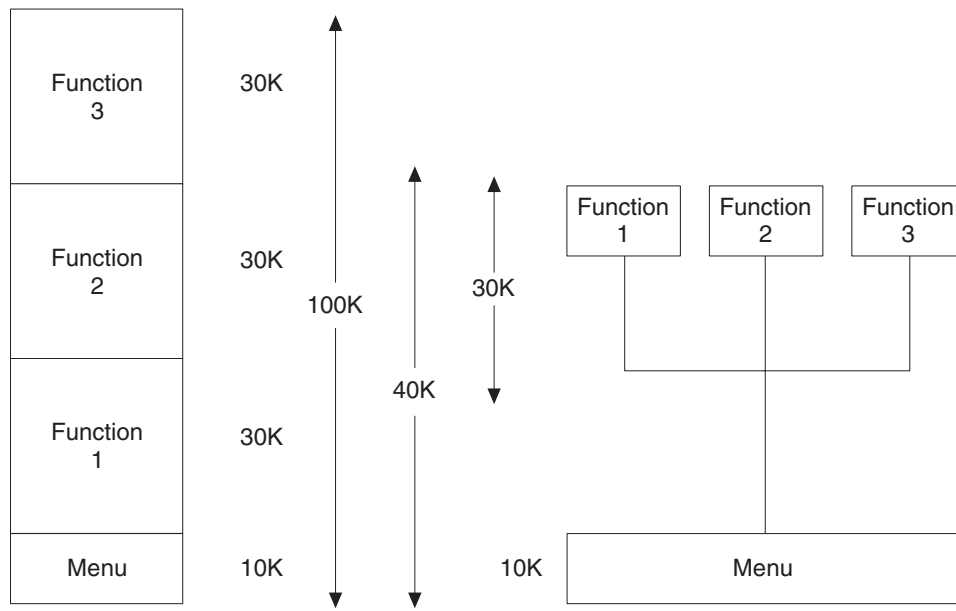


Figure 14. Using Overlays in a Large Program

You can also create nested overlays in the form of a tree structure. Figure 15 on page 217 shows a tree structure overlay.

The top of the highest overlay determines the total amount of memory required. In Figure 15 on page 217, the highest overlay is SUB4. This overlay requires substantially less memory than would be required if all the functions and subfunctions were to reside in memory simultaneously.

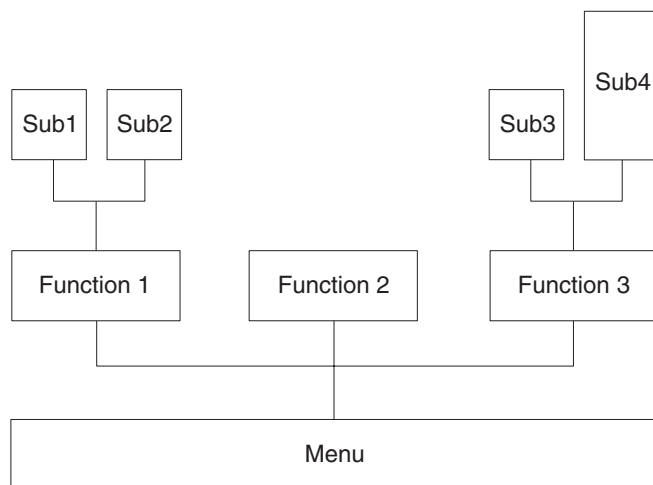


Figure 15. Tree Structure of Overlays

## Overlay Command Line Syntax

You specify overlays in the LINK86 command line by enclosing each overlay specification in parentheses.

You can specify an overlay in one of the following formats, where ROOT.OBJ is the object file that calls the overlays:

```
A:>LINK86 ROOT (OVERLAY1)
A:>LINK86 ROOT (OVERLAY1,PART2,PART3)
A:>LINK86 ROOT (OVERLAY1=PART1,PART2,PART3)
```

- The first form produces the files ROOT.286 and OVERLAY1.OVR from ROOT.OBJ and OVERLAY1.OBJ.
- The second form produces the files ROOT.286 and OVERLAY1.OVR from ROOT.OBJ, OVERLAY1.OBJ, PART2.OBJ, and PART3.OBJ.
- The third form produces the files ROOT.286 and OVERLAY1.OVR from ROOT.OBJ, PART1.OBJ, PART2.OBJ, and PART3.OBJ.

On the command line, a left parenthesis indicates the start of a new overlay specification and the end of the previous overlay specification. You can use spaces to improve readability, and commas can separate parts of a single overlay. However, do not use commas to separate the overlay specifications from the root module or from each other.

In the following example, the command line is not valid:

```
A:>LINK86 ROOT(OVERLAY1),MOREROOT
```

The correct command line is:

```
A:>LINK86 ROOT,MOREROOT(OVERLAY1)
```

To nest overlays, you must specify them in the command line with nested parentheses.

In the following example, the command line creates the overlay system shown in Figure 15 on page 217:

```
A:>LINK86 MENU(FUNC1(SUB1)(SUB2))(FUNC2)(FUNC3(SUB3)(SUB4))
```

When linking files to be overlayed along with files not to be overlayed, the files to be overlayed are specified last. When you create the file ROOT.286 from the files ROOT.OBJ, PARTA.OBJ, and PARTB.OBJ, to link OVER1.OBJ and OVER2.OBJ as overlays, enter the following command line:

```
A:>LINK86 ROOT,PARTA,PARTB(OVER1, OVER2)
```

---

## The POSTLINK Utility

The POSTLINK Utility is a postprocessor program that converts a LINK86 load module into a module that can be loaded faster and use memory more efficiently in a multitasking environment. Using POSTLINK is mandatory for all load modules that have been linked with the CODESHARED option.

When POSTLINK is invoked, a temporary file called FILENAME.PST is created. When POSTLINK has completed converting the LINK86 load module into a new load module, the temporary file is erased.

### Notes:

1. The POSTLINK Utility must not be used on overlay files or the root module of an overlay file.
2. A Postlinked load module that contains large unreferenced object modules can cause unpredictable results when loaded on a 4683 point-of-sale terminal.

## Invoking the POSTLINK Utility

You invoke the POSTLINK Utility with the following command line format:

```
POSTLINK filespec
```

*filespec* is a file specification of a load module consisting of an optional path specification and a filename with an extension.

The output file has the same filespec as the input file.

```
A:>POSTLINK C:\LOADMODS\TEST.286
```



## Use of ERRORLEVEL Test

When running the POSTLINK Utility from a batch file, the POSTLINK operation can be tested for errors. If POSTLINK is successful, an ERRORLEVEL value of zero is returned. To test for an error, use the batch file statement IF ERRORLEVEL 1...

The following is an example within a batch file:

```
POSTLINK TEST.286  
IF ERRORLEVEL 1 ECHO We have a problem >> RESULTS
```

In the above example, if the POSTLINK is unsuccessful the message “We have a problem” is written to the file RESULTS.



---

## Chapter 10. Using the Print Spooler Utility

The operating system has a Print Spooler Facility that enables you to send your files to one of eight queues for printing on one of eight printers. This spooling facility can be shared by concurrently executing applications.

System configuration allows a maximum of eight printers to be defined. Printers are assigned numbers (PRN1: through PRN8:) at configuration time. An application program uses this number to gain access to a particular printer. An application can gain access to the printer assigned to the console the application is running on by using the printer name PRN:.

One of the printers (PRN1: through PRN8:) can also be defined as the system printer. Any application can access the system printer using PRN0:.

**Note:** By using the application service ADXSERVE, applications can determine the printer assigned to the console it is running on.

Your application must open the printer in UNLOCKED mode. This mode allows multiple applications to use the printer. The printer must be closed before printing can be scheduled. You can use either CLOSE or TCLOSE. CLOSE releases the application from all control of the file; TCLOSE allows limited control.

If your application uses TCLOSE, the application can receive job or printer status information. The TCLOSE is performed when the application completes sending data to the print spooler. The file is then scheduled for print. The print spooler allows only reading of a job's status. If the application decides some action needs to be taken, it can issue special commands to perform the action on a print job or queue. See "Using Special Commands" on page 223 for more information on these special commands.

The job status is obtained by the application performing a read after a TCLOSE. The spooler returns job and queue status as detailed in the next section. Special commands are issued by performing a WRITE FORM with a string of characters representing the appropriate command.

---

### Obtaining Job Status after a TCLOSE

To obtain a job's status, your application must perform a read with a buffer capable of holding at least 16 characters. The spooler places the job's status in the buffer.

The first seven characters in the buffer correspond to specific events.

If any of these seven characters is occupied by a space, that particular event did not occur since the last read was performed. An event that has occurred since the last read is marked by an asterisk. The characters indicate the following events:

- Character 0  
Job completed. The job has been entirely despoiled to the printer and removed from the system.
- Character 1  
Job canceled. The job was removed from the system before it could be printed. The user either canceled that particular job or an authorized user purged an entire print queue.
- Character 2  
Printer held. A hold command has been issued, and no jobs are currently being despoiled from that particular queue. When an activate command is issued, the job currently at the top of the queue starts over.
- Character 3  
Printer error. The printer has detected an error in printing. This does not include errors such as out-of-paper, powered off, and so on.

- Character 4  
Out-of-Paper. The printer has run out of paper and is currently waiting.
- Character 5  
Printer timeout. The printer has taken too long to give the “go ahead” signal.
- Character 6  
Printer powered off. The printer lost power. This event is serious because it involves a loss of data.

The last nine characters are used as follows:

- Character 7  
Indicates the printer the job was queued for. This value is a number 1 through 8. If the job is being held (not the queue), this character is an asterisk.
- Characters 8 through 10  
Indicate the job’s current position for printing. ‘000’ indicates the job is currently being despoiled to the printer.
- Characters 11 through 13  
Indicate the job’s current job ID.
- Characters 14 and 15  
Are carriage return and line feed characters, which indicate the end of the record.

The only error that is returned from a read is an implementation error. (This type of error is defined in “Error Return Codes for the Print Spooler” on page 225.) This error is returned if a read is attempted and the job has not been closed using TCLOSE.

---

## Issuing a Command to the Print Spooler

Following a TCLOSE, an application can issue a special command that affects a job or a queue. Special commands are issued by writing all necessary data to perform the action.

The format for these special commands is the following (number in parentheses is the number of characters for that variable):

*command*(2),*jobid*(3),*source*(1),*destination*(1),*node*(8),

where:

*command* =

- PJ** = Move priority job to top of queue.
- TJ** = Transfer job to another queue.
- CJ** = Cancel the job.
- HJ** = Hold the job.
- AJ** = Activate the job.
- HQ** = Hold the queue.
- AQ** = Activate the queue.
- LQ** = Load the queue following an IPL.
- TQ** = Transfer a queue.
- RQ** = Resume a queue after previous transfer.
- CQ** = Cancel a queue.

*jobid* =

The three-character variable indicates the job to be affected (commands ending with a J). Default job is the job corresponding to this OPEN.

*source* =

The one-character variable indicating the queue to be affected (commands ending with a Q). Valid values are 1 to 8. Default queue is the one corresponding to this OPEN.

*destination* =

The queue to receive jobs (commands TJ and TQ). Valid values are 1 to 8. This variable has no default.

*node* =

Eight-character variable used for commands TJ and TQ. If the destination printer is across the network, this variable supplies the node name.

All parameters are optional depending on the command and whether or not a default exists. A delimiting comma must replace each parameter not entered. For example, if an application wanted to terminate one of its own jobs, the following command would terminate job 001 corresponding to the I/O session number used:

CJ,001,,, ,

The following command transfers all current and future files for PRN1: to PRN3:.

TQ,,1,3,, ,

The following command results in job 073 being transferred to the PRN5: printer at node DD (if such a printer exists).

TJ,073,,5,ADXLXDDN

This command cancels all jobs currently queued (and not being held) for PRN4:.

CQ,,4,,, ,

---

## Using Special Commands

The following nine special commands are available to applications that have performed a TCLOSE:

- PJ (Move priority job to top of queue)

An application can perform this operation on a job that is currently queued (but not being held). This command moves the job to the position immediately following the current job in whatever queue it is residing. If the job is currently printing or is the next job to print, the operation completes successfully. Possible errors returned from this operation include Job Does Not Exist and Queue Full.

PJ,001,1,,,  
└──┬── source  
   └── job ID

- TJ (Transfer job to another queue)

An application can perform this operation on any job that has been scheduled to print, regardless of whether it is currently queued or being held. The command moves the job from wherever it is to the destination printer specified and queues it for that printer. Therefore a job could be moved from one queue to another or activated with this command. The destination printer might be across the network as long as a proper node and printer are specified. Network moves are done by writing the job's file across the network. Non-network moves are done by simply moving the job's record from one queue to another. Network moves, therefore, take more time to complete than a non-network move.

TJ,001,1,1,ADXLXCCN

```

graph LR
    TJ[TJ,001,1,1,ADXLXCCN]
    TJ --> jobID[job ID]
    TJ --> source[source]
    TJ --> destination[destination]
    TJ --> node[node]
  
```

- CJ (Cancel a job)

This operation can be carried out on any job scheduled to print, regardless of whether it is currently queued or being held. The job entry is deleted from the appropriate queue and the job's file is erased. No recovery is possible.

CJ,001,1,,,

```

graph LR
    CJ[CJ,001,1,,,]
    CJ --> jobID[job ID]
    CJ --> source[source]
  
```

- HJ (Hold a job)

Any job that is currently queued can be placed on hold. If an attempt is made to place an already held job on hold, a successful return code is received. Jobs placed on hold are held indefinitely. However, a limit of 32 held jobs exists. Any attempt to hold more than this receives a queue full error.

HJ,001,1,,,

```

graph LR
    HJ[HJ,001,1,,,]
    HJ --> jobID[job ID]
    HJ --> source[source]
  
```

- AJ (Activate a job)

This command can be used on any held job. It is activated in the queue from which it was held. If an alternate queue is wanted, then the move command should be used.

AJ,001,1,,,

```

graph LR
    AJ[AJ,001,1,,,]
    AJ --> jobID[job ID]
    AJ --> source[source]
  
```

- HQ (Hold the queue)

To hold all jobs scheduled to print, issue the HQ command. This command places the queue on hold indefinitely.

- AQ (Activate a held queue)

This command allows a held print queue to resume printing the jobs.

AQ,,1,,,

```

graph LR
    AQ[AQ,,1,,,]
    AQ --> source[source]
  
```

- LQ (Load a queue)

This command is used only in the special instance where an IPL has occurred. If the spooler detects unfinished jobs in a queue after an IPL, it enters recovery mode. In this mode, no affected queues are restarted until a load queue command is given. However, a queue is automatically restarted after an IPL if it is sent a new job to queue.

LQ,,1,,,

```

graph LR
    LQ[LQ,,1,,,]
    LQ --> source[source]
  
```

**Note:** Only an authorized user with an access of Group 2 - User 1 or higher can perform the following commands (transfer, resume, and cancel queue).

- TQ (Transfer a queue)

When a printer needs to be taken offline, it is possible to transfer all of its jobs to an alternate printer. This alternate printer must be within the system or within the network. Transfers within the system are checked for situations where, for example, PRN2: is redirected to PRN1: and PRN1: is redirected to PRN2:. Transfers across the network are not checked, and it is the user's responsibility to prevent a loop. All transfers remain active.

TQ,,1,1,ADXLCCN

node  
destination  
source

- RQ (Resume a queue)

After a printer has been transferred, you can place it back online using this command.

To ensure that the WRITE FORM statement goes to the correct print spooler driver, open the device SPRN1: on the controller that the application is running. Otherwise, if your PRN1: is transferred to another controller your WRITE FORM might receive an implement error (80894009).

RQ,,1,,,

source

- CQ (Cancel all the jobs in a queue)

Use this command if you need to cancel all of the jobs scheduled to print on a particular printer. All jobs currently queued for the specified printer are removed from the queue and their corresponding files deleted. Jobs that have been held from this queue are not affected.

CQ,,1,,,

source

---

## Error Return Codes for the Print Spooler

If a write of a special command is attempted before a TCLOSE has been performed, no error is returned. The command is treated like print data and is spooled with all other data.

If a proper special command is written to the spooler following a TCLOSE, the following error return codes are possible:

- JOB HELD (80890006)

For a PJ command, this means the job is not in a queue and needs to be activated first. For an HJ command, it means that because of a system failure, the held job has no printer associated with it and to be activated, a transfer command needs to be used.

- NOJOB (80890002)

The job ID specified is not currently in use in the system (the job was finished or canceled).

- LOOP (80890003)

The requested redirection would result in a loop because of previous transfers.

- QUEUEFULL (80890004)

The requested action could not be completed because of a full queue.

- AUTHORIZE (80890005)

The requested action is not possible because the requestor is not operating under a group-user that allows such actions.

- **IMPLEMENT (80894009)**

A command is either missing a required parameter, contains a parameter that is not valid, or was issued by an unauthorized user.

In addition, any errors that might result from trying to open a printer can also be returned (when performing a network move or a network redirection.)

When an error occurs, the print spooler handles each of the printer errors in the same way. If any other action is wanted, you must initiate it.

When the spooler detects an error, it sets up a timer lasting 30 seconds. Upon completion of the timer, the spooler resumes where it left off and continues trying to send data to the printer. For the timeout error, the print drivers handle the time. Therefore, the spooler does not start a timer; it continues trying to send data.

The type of reaction to these errors should depend on how serious a situation is considered to be. While timeout and paper out do not cause the loss of data and might not be a serious error, a printer error can cause the loss of data, which can be serious.

If an application detects these errors, a reasonable response would be to print a message for you indicating something is wrong. If the error involves certain data loss, it would even be reasonable to issue a command to hold the queue. A hold causes the current job to be restarted from the beginning when the queue is activated thus ensuring it is printed in its entirety.

However, if the printer is connected through serial interface, many (possibly all, depending on the interface and communication scheme) of the errors are detectable only as timeout errors. Therefore, there would be no way to distinguish between the errors.

**Note:** If the same application is to perform both reads and writes of a special command after TCLOSE, the reads and writes must be performed under different I/O session numbers.



---

## Chapter 11. Using the Disk Surface Analysis Utility

---

### Introduction to the Disk Surface Analysis Utility

**Note:** The Disk Surface Analysis Utility is supported only in Classic mode.

The Disk Surface Analysis Utility is designed to minimize the disruption to the store while recovering from disk surface failures. Previously, the procedure used for recovering from hard disk surface failure involved manually formatting or replacing the hard disk. The operating system and other data would have to be reloaded on the new hard disk.

Using the Disk Surface Analysis Utility allows you to recover from surface failures by replacing only the data affected by the surface failure. You do not need to replace or format the hard disk. If host communications are available, an operator can perform the disk surface analysis remotely at the host site.

The IPL Command Processor provides the capability to execute a sequence of commands during the store controller IPL. The IPL Command Processor is necessary to start the Disk Surface Analysis Utility from the host. See Chapter 12, “Using the IPL Command Processor,” on page 239 for more information about the IPL Command Processor.

---

### Disk Surface Analysis Utility

The Disk Surface Analysis Utility performs a surface analysis on a hard disk. The utility can also relocate files away from areas containing surface defects on the hard disk. The defective surface areas are marked in the file allocation table to prevent future access to these areas. If the defective area is allocated to a file, some data is lost. To replace the damaged file, you must copy the data from another source.

For each defective surface area, ADXCW0L reports the file name or subdirectory name that is allocated in the defective disk space. Unallocated areas with surface defects are also reported.

You can use the following formats to invoke the Disk Surface Analysis Utility. These formats are explained in “Command Formats for ADXCW0L” on page 228.

- ADXCW0L drive:
- ADXCW0L drive:\filename.filetype
- ADXCW0L drive: -parameters
- ADXCW0L drive:\filename.filetype -parameters

Subdirectories are reported as being defective, but are not fixed. Information appears that includes the defective cluster number, the subdirectory name, and a brief message indicating that the correction has not been performed.

After invoking a Disk Surface Analysis Utility command, two sections of information appear. The first section displays the defective cluster information. The information can be a sector number specified by a user or reported by the utility. The surface analysis information includes the following:

- Defective cluster number
- The status of the disk space
  - Allocated
  - Unallocated
- Cylinder head sector address
- Relative sector number
- Error code (returned during the read-verify of the sector)

**Note:** When the sector parameter (-R) is specified, the information appears regardless of the error code. See "Parameter Descriptions for ADXCSW0L" on page 228 for more information on the sector parameter.

The second part of the report lists the file names that are relocated. The following information is displayed:

- New cluster number that contains the recovered data
- The defective cluster number that was replaced
- The name of the file
- The offset and size of the data area that was defective

You can invoke the Disk Surface Analysis Utility without the -F parameter to report the current defects and changes that have been made.

---

## Parameter Descriptions for ADXCSW0L

Descriptions of the parameters provided with the ADXCSW0L routine are listed below.

**-R** Allows you to specify a defective sector on a hard disk. It is referred to as the sector parameter. This parameter must be followed by a decimal or hexadecimal (prefixed by 0X) number, which represents a relative sector number. The first sector on a hard disk is the relative sector number 0. ADXCSW0L assumes that the disk sector that is specified by the -R parameter is defective. The cluster corresponding to the specified sector is marked defective in the file allocation table. The data is copied to a new area on the disk when the -F parameter is specified.

**Note:** If you do not know if the sector is defective, invoke the utility without the -F parameter. All information about the sector is printed. The data remains in place.

**-F** Allows you to make repairs. If you do not specify this parameter, the information is listed, but not corrected. Unallocated areas with surface defects are marked defective in the file allocation table. Each cluster that is allocated to a file that contains defective sectors is remapped to another cluster. The former cluster is marked defective in the file allocation table. The data is read into the new cluster, and the sectors that cannot be read are replaced with fill data. The default fill character is 0x2E.

**Note:** The file allocation table and the root directory cannot be remapped. Errors within these areas can be reported, but cannot be corrected.

**-C** Allows you to specify the fill character to replace the unreadable sectors in a file. The -C parameter can override the default fill character.

---

## Command Formats for ADXCSW0L

ADXCSW0L *drive*: performs a surface analysis of the entire disk drive. All surface defects that are not flagged in the FAT are reported, and the information on the repaired files is displayed. If the -F parameter was specified, the default character 'X'2E' is used when the data is recovered from the defective area of the hard disk.

**Note:** If the -F parameter is not specified, a message is displayed on the screen that indicates defects are reported but not corrected.

### Example 1

The following command scans the entire C: drive. Surface defects are listed but not corrected. The output report is listed below the command.

```
C>ADXCSW0L C:
  Surface defects are reported but not fixed.

  Attempting to recover C:
```

```

*****
****   Surface Defects   ****
*****

Cluster Number:      0032
Status:              Allocated
Cylinder:            4
Head:                1
Sector:              8
Relative Sector Number: 16c
Error Code:          86210004

```

```

*****
****   Files Repaired   ****
*****

```

```

Cluster Number:      0033
Replaces Cluster Number: 0032
File Name:           C:\TEST1.DAT
Lost Data Offset:     Lost Data Size:
3500                  0200

```

ADXCSW0L *drive:filename.filetype* performs a surface analysis on the disk space occupied by the specified file. If an error is located, the defect is reported. The fill character defaults to X'2E'.

**Note:** The correction is made if the -F parameter is specified.

## Example 2

The following command recovers the file TEST2.DAT on the D: drive. Surface defects are reported but not corrected. The output report is listed below the command.

```

C>ADXCW0L D:\TEST2.DAT
  Surface defects are reported but not fixed.

```

```

Attempting to recover D:\TEST2.DAT
*****
****   Surface Defects   ****
*****

Cluster Number:      0946
Status:              Allocated
Cylinder:            071
Head:                03
Sector:              06
Relative Sector Number: 25BD
Error Code:          86210004

```

```

*****
****   Files Repaired   ****
*****

```

```

Cluster Number:      0033
Replaces Cluster Number: 0946
File Name:           C:\TEST2.DAT
Lost Data Offset:     Lost Data Size
3500                  0200

```

ADXCW0L *drive: -parameters* allows any combination of the parameters. However, a sector number cannot be specified when a file name is specified.

## Example 3

In this example, the command uses the sector parameter, the fill character parameter, and the -F parameter. The specified sector parameter is assumed to be defective. All of the data on the defective

sector is relocated. The specified fill character is used in the reallocated disk space. All changes are written to the disk because the -F parameter is issued. The output report is listed after the command.

```
C>ADXCSW0L D: -R0x25D1 -C0x3D -F
Attempting to recover D:

With specified sector number 25D1.
Information on specified bad sector:
Cluster Number:      094B
Status:              Allocated
Cylinder:            071
Head:                04
Sector:              09
Relative Sector Number: 25D1
Error Code:          86210004

*****
****   Files Repaired   ****
*****
Cluster Number:      0950
Replaces Bad Customer: 094B
File Name:           TEST3.DAT
Lost Data Offset:    Lost Data Sizes
1800                  200
```

ADXCSW0L *drive:\filespec -parameters* performs a surface analysis on a specified file name to be specified along with the parameters. However, only the fill character and the fix parameters are allowed with the file name. The sector parameter -R is not allowed. A message appears if the -R is included in the command.

## Example 4

The following command recovers the file TEST4.DAT on the D: drive. The -F and -C parameters allow the surface defect areas to be corrected and filled with the fill character 0x4A. The output report is listed below the command.

```
ADXCSW0L D:\TEST4.DAT -C0x4A -F
Attempting to recover D:\TEST4.DAT

*****
****   Surface Defects   ****
*****

Cluster Number:      0947
Status:              Allocated
Cylinder:            071
Head:                03
Sector:              09
Relative Sector Number: 25C0
Error Code:          86210004

*****
****   Files Repaired   ****
*****

Cluster Number:      0046
Replaces Cluster Number: 0947
File Name:           C:\TEST4.DAT
Lost Data Offset:    Lost Data Sizes
2600                  0200
```

---

## Using the Disk Surface Analysis Utility to Recover Data

Use the Disk Surface Analysis Utility to locate surface defects and to mark them defective in the FAT. Data may be moved if necessary. This process prevents future access to the defective areas on the hard disk. If the defective area is within a file, some data is lost. Replace the damaged file by copying the file from another source.

The store controller might dump, or a power line disturbance might occur during an IPL. The command file should be organized so that problems are not caused by running the same command multiple times. For more information on command files, see “Example of a Command File” on page 240.

To ensure that the command file executes only one time, place an ERASE command at the end of the command file. Perform the erase at time frame 3 to distribute the request.

If you use RCP to re-IPL, and you configure it to start at each IPL as a background application, the IPL Command Processor command file ADXILIOF.DAT should contain a time frame 3 command to erase the RCP selection file ADXCSCF.DAT. The ERASE command prevents a store controller IPL loop.

If a surface defect prevents the IPL of a store controller, or if host communications cannot be established, the Disk Surface Analysis Utility can be invoked from the Supplemental Diskettes or the Supplemental Option using the CD-ROM. You can use the disk rebuild function to obtain specific replacement files from another store controller on the LAN (MCF Network). In a non-LAN environment, you must obtain the replacement files from a diskette or tape.

The following steps explain how to use the Disk Surface Analysis Utility for disk recovery.

**Note:** You can omit steps 1 through 3—which contain the initial disk analysis command, re-IPL, and report sequence—if you know a particular file is damaged (for example, an error message has been logged on a file).

1. Transfer the command file containing the ADXCW0L command from the host. The command requests that the Disk Surface Analysis Utility be started on the store controller's hard disk.
2. IPL the store controller manually or by using the RCP re-IPL command. See the *4690 OS: Communications Programming Reference* for more information.
3. The store controller generates the utility's output report, and the host retrieves it. The report indicates hard disk areas that are defective.
4. Send a new command file to the store controller containing the following commands:
  - Disk Surface Analysis Utility command using the -F parameter.
  - ERASE or RENAME command of files containing surface defects. You can use the RENAME command to save a copy of the file that contains unreadable data replaced by fill characters.
  - COPY command for each file with surface defects from any store controller on the LAN. If the store controller is not on a LAN, the file can be transmitted from the host.
5. IPL the store controller and the repairs are performed.
6. The store controller generates a second output report at the store controller, and the host retrieves it to verify the changes.
7. Send a default (empty) command file to the store.

**Note:** To ensure that a command file can be transmitted to the store after a surface defect appears, you can send a default command file containing zeros or blanks equaling the maximum number of commands that you need. Use the HCP LOAD FILE command or the RCMS SEND command to overwrite the existing file with the new command file.

---

## Using the Time Frame Indicators

As part of the disk recovery procedure, you must execute some commands within a specified time frame. The time frame default is -3. For example:

<b>CHKDSK</b>	-1
<b>ADXCW0L</b>	-1
<b>RENAME</b>	-2
<b>COPY</b>	-2
<b>COMMAND -c ERASE</b>	-2

RENAME and COPY are executed within time frame -2, so the operation is not distributed. If a RENAME, COPY, or ERASE in the command file is to be distributed, the command should be preceded by -3.

You should use time frame -1 only for executing CHKDSK and the Disk Surface Analysis Utility. Use time frame -2 only for renaming, erasing, or copying to files that have been repaired by the Disk Surface Analysis Utility.

See Chapter 12, "Using the IPL Command Processor," on page 239 for more information on time frame indicators.

---

## Case Examples of Disk Recovery

This section describes some example scenarios and disk recovery procedures. In an actual situation, multiple files can be affected, a situation which would require you to develop a recovery procedure. However, in these examples, only one file is affected by a surface defect.

### Non-LAN (MCF Network), Defect in .286 File

**Symptom:** A user in a non-LAN environment cannot execute the Format Dump Data Utility (ADXCWL0L.286).

The following messages were logged:

- A W754 B4/S004/E021 with RC=80210004 or RC=8021000D
- A W754 B4/S004/E018 with the file name: ADXCWL0L.286

#### Action:

1. Create the following command file at the host site and transmit it to the store as ADX\_SDT1:ADXILIOF.DAT

```
-1 adxcw0l c:
```
2. Execute the following RCP command to re-IPL the store controller:

```
adxc20l N 13
```
3. Retrieve ADX\_SDT1:ADXNSCCF.DAT (assumes store controller node is CC) from the store. Verify that the only file with a surface defect is ADXCWL0L.286.
4. Transmit a new copy of ADXCWL0L.286 to subdirectory ADX\_SMNT.
5. Create the following command file at the host site and transmit it to the store as ADX\_SDT1:ADXILIOF.DAT

```
-x -1 adxcw0l c: -f
command -c erase ADX_SPM:ADXCWL0L.286
rename ADX_SMNT:ADXCWL0L.286 ADX_SPM:ADXCWL0L.286
```
6. Execute the following RCP command to re-IPL the store controller:

```
adxc20l N 13
```

7. Retrieve ADX\_SDT1:ADXNSCCF.DAT (assumes store controller node is CC) from the store. Verify that recovery was successful.
8. Transmit a copy of ADX\_SDT1:ADXILIOF.DAT, which contains no commands to the store.
9. Delete ADX\_SDT1:ADXNSCCF.DAT.
10. If a different version of ADXCSL0L.286 was in the ADX\_SMNT subdirectory prior to this recovery procedure, re-transmit that version to the store.

## LAN (MCF Network), Defect in .286 File on Master Controller

**Symptom:** A user in a two-controller LAN store environment cannot execute the Format Dump Data Utility (ADXCSL0L.286) on the master store controller. The master is node CC and the alternate master is node DD.

The following return codes are logged:

- A W754 B4/S004/E021 with RC=80210004 or RC=8021000D
- A W754 B4/S004/E018 with file name: ADXCSL0L.286

### Action:

1. Create the following command file at the host site and transmit it to the store as ADX\_SDT1:ADXILIOF.DAT
 

```
-1 -nCC adxcsw01 c:
```
2. Execute the following RCP command on the master store controller to re-IPL:
 

```
adxcs201 N 13
```
3. Retrieve ADX\_SDT1:ADXNSCCF.DAT from the store. Verify that the only file with a surface defect is ADXCSL0L.286.
4. Create the following command file at the host site and transmit it to the store as ADX\_SDT1:ADXILIOF.DAT
 

```
-ncc -x -1 adxcsw01 c: -f
-ncc -2 copy adx\yddn::adx_spgm:adxcsl0l.286      gadx_sp m:adxcsl0l.286
```
5. Execute the following RCP command on the master store controller to re-IPL.
 

```
adxcs201 N 13
```
6. Retrieve ADX\_SDT1:ADXNSCCF.DAT from the store. Verify that recovery was successful.
7. Transmit a copy of ADX\_SDT1:ADXILIOF.DAT that contains no commands to the store.
8. Delete ADX\_SDT1:ADXNSCCF.DAT.

## LAN (MCF Network), Defect in .286 File on Alternate Master Controller

**Symptom:** A user in a two-controller LAN environment cannot execute the Format Dump Data Utility (ADXCSL0L.286) on the alternate master. The master is node CC and the alternate master is node DD.

The following return codes are logged:

- A W754 B4/S004/E021 with RC=80210004 or RC=8021000D
- A W754 B4/S004/E018 with file name: ADXCSL0L.286

### Action:

1. Create the following command file at the host site and transmit it to the store as ADX\_SDT1:ADXILIOF.DAT
 

```
-1 -ndd adxcsw01 c:
```
2. Execute the following RCP command to re-IPL the alternate master.
 

```
dd::adxcs201 N 13
```
3. Retrieve ADX\_SDT1:ADXNSDDF.DAT from the store. Verify that the only file with a surface defect is ADXCSL0L.286.

4. Create the following command file at the host site and transmit it to the store as  
 ADX\_SDT1:ADXILI0F.DAT  

```
-nadd -x -1 adxcsw01 c: -f
-nadd -2 copy adx1xccn::adx_spgm:adxcsl0L.286      gadx_sp m:adxcsl0L.286
```
5. Execute the following RCP command to re-IPL the alternate master.  
 dd::adxcsl201 N 13
6. Retrieve ADX\_SDT1:ADXNSDDF.DAT from the store. Verify that recovery was successful.
7. Transmit a copy of ADX\_SDT1:ADXILI0F.DAT that contains no commands to the store.
8. Delete ADX\_SDT1:ADXNSDDF.DAT.

## Non-LAN (MCF Network), Defect in Unallocated Space

**Symptom:** A user in a non-LAN environment cannot transmit a file from the host to the store. The problem could be caused by an unallocated area of the disk that contains a surface defect. Assume that the store controller node is CC.

The following return codes are logged:

- W754 B4/S004/E021 with RC=80210004 or RC=8021000D
- W754 B4/S004/E018 with the name of the file being transmitted

### Action:

1. Create the following command file at the host site and transmit it to the store as  
 ADX\_SDT1:ADXILI0F.DAT  

```
-1 adxcsw01 c: -f
```
2. Execute the following RCP command to re-IPL the store controller:  
 adxcsl201 N 13
3. Retrieve ADX\_SDT1:ADXNSCFF.DAT from the store. Verify that recovery was successful. The error is associated with an allocated cluster of the transmitted file.
4. Transmit a copy of ADX\_SDT1:ADXILI0F.DAT that contains no commands to the store.
5. Delete ADX\_SDT1:ADXNSCCF.DAT.
6. Retransmit the original file to the store.

## LAN (MCF Network), Defect in Unallocated Space on Master Controller

**Symptom:** A user in a two-controller LAN environment was unable to transmit a file from the host to the store. The symptom may be caused by an allocated area of the disk that contains a surface defect. The master is node CC and the alternate master is node DD.

The following return codes are received:

- A W754 B4/S004/E021 with RC=80210004 or RC=8021000D
- A W754 B4/S004/E018 with the name of the file being transmitted

### Action:

1. Create the following command file at the host site and transmit it to the store as  
 ADX\_SDT1:ADXILI0F.DAT  

```
-1 -ncc adxcsw01 c: -f
```
2. Execute the following RCP command on the master to re-IPL it:  
 adxcsl201 N 13
3. Retrieve ADX\_SDT1:ADXNSCCF.DAT from the store. Verify that the recovery was successful. The error is associated with an allocated cluster of the transmitted file.
4. Transmit a copy of ADX\_SDT1:ADXILI0F.DAT that contains no commands to the store.
5. Delete ADX\_SDT1:ADXNSCCF.DAT.



6. Retransmit the original file to the store.

## LAN (MCF Network), Defect in Unallocated Space on Alternate Master Controller

**Symptom:** A two-controller LAN store environment cannot transmit a file from the host to the store. A Distribution Exception Log entry was logged for the file being transmitted at node DD. The master is node CC and the alternate master is node DD.

Message W754 B4/S004/E021 with RC=80210004 or RC=8021000D was logged at node DD.

### Action:

1. Create the following command file at the host site and transmit it to the store as  
ADX\_SDT1:ADXILI0F.DAT  
-1 -nnd adxcsw01 c: -f
2. Execute the following RCP command to re-IPL the alternate master:  
dd::adxcsw01 N 13
3. Retrieve ADX\_SDT1:ADXNSDDF.DAT from the store. Verify that the recovery was successful. The file that was transmitted from the host is reconciled during the IPL of the alternate master.
4. Transmit a copy of ADX\_SDT1:ADXILI0F.DAT that contains no commands to the store.
5. Delete ADX\_SDT1:ADXNSDDF.DAT.

## Non-LAN (MCF Network), Defect in Item Record File

**Symptom:** The user is in a non-LAN environment. The item record file on the D: drive contains a defect.

The following errors were logged:

- W754 B4/S004/E021 with RC=80210004 or RC=8021000D
- W754 B4/S004/E018 with file name EALITEMR.DAT

### Action:

1. Create the following command file at the host site and transmit it to the store as  
ADX\_SDT1:ADXILI0F.DAT  
-1 adxcsw01 d:
2. Execute the following RCP command to re-IPL the store controller:  
adxcsw01 N 13
3. Retrieve ADX\_SDT1:ADXNSCCF.DAT (assumes store controller node is CC) from the store. Verify that the only file with a surface defect is EALITEMR.DAT.
4. Create the following command file at the host site and transmit it to the store as  
ADX\_SDT1:ADXILI0F.DAT  
-1 adxcsw01 d: -f -c0

**Note:** A fill character of binary 0 is recommended for use when recovering a keyed file.

5. Execute the following RCP command to re-IPL the store controller:  
adxcsw01 N 13
6. Retrieve ADX\_SDT1:ADXNSCCF.DAT (assumes store controller node is CC) from the store. Verify that recovery was successful.
7. Transmit a copy of ADX\_SDT1:ADXILI0F.DAT that contains no commands to the store.
8. Delete ADX\_SDT1:ADXNSCCF.DAT.
9. Use existing procedures to reload the item record file. Until this is done, attempts to access items that were present in the damaged section of the item record file fails.

## LAN (MCF Network), Defect in Item Record File on Master Controller

**Symptom:** In a two-controller LAN environment, the master is node CC and the alternate master is node DD. The item record file is on the D: drive, which does not contain enough free space on the D: drive for a second copy of it.

The following return codes were logged against the item record file on the master:

- W754 B4/S004/E021 with RC=80210004 or RC=8021000D
- W754 B4/S004/E018 with file name: EALITEMR.DAT

### Action:

1. Create the following command file at the host site and transmit it to the store as ADX\_SDT1:ADXILIOF.DAT  

```
-ncc -1 adxcsw01 d:
```
2. Execute the following RCP command on the master to re-IPL it:  

```
adxcs201 N 13
```
3. Retrieve ADX\_SDT1:ADXNSCCF.DAT from the store. Verify that the only file with a surface defect is EALITEMR.DAT.
4. Create the following command file at the host site and transmit it to the store as ADX\_SDT1:ADXILIOF.DAT.  

```
-1 -ncc -x adxcsw01 d: -f -c0  
-2 -ncc command -c erase d:adx_idt1\ealitemr.dat  
-2 -ncc copy adx\ddn::d:\adx_idt1\ealitemr.dat &d:\adx us.idt1\ealitemr.dat
```
5. Execute the following RCP command on the master to re-IPL it:  

```
adxcs201 N 13
```
6. Retrieve ADX\_SDT1:ADXNSCCF.DAT from the store. Verify that recovery was successful.
7. Transmit a copy of ADX\_SDT1:ADXILIOF.DAT that contains no commands to the store.
8. Delete ADX\_SDT1:ADXNSCCF.DAT.

## LAN (MCF Network), Defect in Item Record File on Alternate Master

**Symptom:** On a two-controller LAN store environment, W754 errors are periodically logged against the item record file on the alternate master. The master is node CC and the alternate master is node DD. The item record file is on the D: drive, and there is not sufficient free space on the D: drive for a second copy of it.

These return codes were logged:

- W754 B4/S004/E021 with RC=80210004 or RC=8021000D
- W754 B4/S004/E018 with file name: EALITEMR.DAT

### Action:

1. Create the following command file at the host site and transmit it to the store as ADX\_SDT1:ADXILIOF.DAT  

```
-ncc -1 adxcsw01 d:
```
2. Execute the following RCP command to re-IPL the alternate master.  

```
dd::adxcs201 N 13
```
3. Retrieve ADX\_SDT1:ADXNSDDF.DAT from the store. Verify that the only file with a surface defect is EALITEMR.DAT.
4. Create the following command file at the host site and transmit it to the store as ADX\_SDT1:ADXILIOF.DAT.  

```
-1 -ncc -x adxcsw01 d: -f -c0  
-2 -ncc command -c erase d:adx_idt1\ealitemr.dat  
-2 -ncc copy adx\ccn::d:\adx_idt1\ealitemr.dat &d:\adx us.idt1\ealitemr.dat
```

5. Execute the following RCP command to re-IPL the alternate master:

```
dd::adxcs201 N 13
```

6. Retrieve ADX\_SDT1:ADXNSDDF.DAT from the store. Verify that the recovery was successful.
7. Transmit a copy of ADX\_SDT1:ADXILI0F.DAT that contains no commands to the store.
8. Delete ADX\_SDT1:ADXNSDDF.DAT.

## Non-LAN (MCF Network), Defect Near End of TLOG

**Symptom:** This example is a variation of a previous example. In a non-LAN store environment, unallocated space is causing errors. The difference is that this problem occurs as the terminals attempt to write to the end of the transaction log.

The following errors were logged as the terminals attempt to write to the transaction log:

- W754 B4/S004/E021 with RC=80210004 or RC=8021000D
- W754 B4/S004/E018 with file name: EALTRANS.DAT

### Action:

1. Create the following command file at the host site and transmit it to the store as ADX\_SDT1:ADXILI0F.DAT

```
-1 adxcsw01 c: -f
```
2. Execute the following RCP command to re-IPL the store controller:

```
adxcs201 N 13
```
3. Retrieve ADX\_SDT1:ADXNSCCF.DAT from the store. Verify that recovery was successful.
4. Transmit a copy of ADX\_SDT1:ADXILI0F.DAT that contains no commands to the store.
5. Delete ADX\_SDT1:ADXNSCCF.DAT.

## LAN (MCF Network), Defect Near End of TLOG on File Server

**Symptom:** A store in a two-controller LAN environment cannot write data to the end of the transaction log because of a defect in unallocated space. The file server/master is CC and the alternate file server/alternate master is DD.

The following errors are logged as the terminals attempt to write to the transaction log:

- W754 B4/S004/E021 with RC=80210004 or RC=8021000D
- W754 B4/S004/E018 with file name: EALTRANS.DAT

### Action:

1. Create the following command file at the host site and transmit it to the store as ADX\_SDT1:ADXILI0F.DAT

```
-1 -ncc adxcsw01 c: -f
```
2. Execute the following RCP command to re-IPL the file server:

```
adxcs201 N 13
```
3. Retrieve ADX\_SDT1:ADXNSCCF.DAT from the store. Verify that recovery was successful.
4. Transmit a copy of ADX\_SDT1:ADXILI0F.DAT that contains no commands to the store.
5. Delete ADX\_SDT1:ADXNSCCF.DAT.

## LAN (MCF Network), Defect Near End of TLOG on Alternate File Server

**Symptom:** In a two-controller LAN environment, the file server node is CC and the alternate file server is DD. The Distribution Exception Log for the file server contains an entry for the transaction log.

The following errors were logged by the alternate file server:

- W754 B4/S004/E021 with RC=80210004 or RC=8021000D

- W754 B4/S004/E018 with file name: EALTRANS.DAT

**Action:**

1. Create the following command file at the host site and transmit it to the store as  
ADX\_SDT1:ADXILI0F.DAT  
-1 -nnd adxcsw01 c: -f
2. Execute the following RCP command to re-IPL the alternate file server:  
dd::adxcsw01 N 13
3. Retrieve ADX\_SDT1:ADXNSDDF.DAT from the store. Verify that the recovery was successful. Reconciliation should update the alternate file server with a current copy of the transaction log.
4. Transmit a copy of ADX\_SDT1:ADXILI0F.DAT that contains no commands to the store.
5. Delete ADX\_SDT1:ADXNSDDF.DAT.

## Chapter 12. Using the IPL Command Processor

The IPL Command Processor provides the capability to execute a sequence of commands during the store controller IPL. The IPL Command Processor is necessary to start the Disk Surface Analysis Utility from the host.

The IPL Command Processor executes commands at three separate times in the IPL sequence. The commands are located in the command file ADXILI0F.DAT, in the subdirectory ADX\_SDT1. You must create the command file. Each command must have a time frame indicator to specify when the processor should invoke the command. This is necessary because certain commands can be performed only at specified times during the IPL sequence.

*Table 22. Using Time Frame Indicators*

Time Frame Indicator	Description
1	Commands that the IPL Command Processor processes early in the IPL sequence occur before the creation of the operator console facility (OCF) error logging process. Commands executed at this point can gain exclusive access to the disk. LAN (MCF Network) services are not available. ADXSERVE and ADXSTART functions are not available. Error logging (ADXERROR) is available, but any messages logged will not be written to the system log file until the IPL completes.
2	Commands that the IPL Command Processor processes in the middle of the IPL sequence occur after the installation of the LAN drivers and before the installation of Data Distribution. Primitive LAN services are available, but Data Distribution is not active to prevent operations on image files. File updates are not distributed. Image versions of distributed files can be updated, renamed, and erased. The operating system does not resolve discrepancies automatically. Prime versions of distributed files can be updated, renamed, and erased, although these changes are not performed on the corresponding image files. Disk repair procedures require changes to the image file, but the changes can cause damage if not used carefully. The user is responsible for the results of all file updates made during time frame 2. Only files that are defective and have been repaired by the Disk Surface Analysis Utility should be modified during time frame 2. ADXSERVE and ADXSTART functions are not available. Error logging (ADXERROR) is available, but any messages logged will not be written to the system log file until the IPL completes.
3	Commands that the IPL Command Processor processes late in the IPL sequence occur after most drivers and services have been initialized. However, ADXSTART is not available.  At each of the three time frames, the IPL Command Processor executes the commands for that phase in the order that they appear in the command file. The only control structure provided by the IPL Command Processor is the Exit On Error option (-x). If a command that has been prefaced by the Exit on Error option ends in an error, the IPL Command Processor executes no other commands for the remainder of the IPL sequence. An error is a non-zero return code.  Each command appears on the screen as it is being executed. Output on the display for each command is redirected to a temporary RAM disk work file. This allows the command to have exclusive access to the disk. The work file is copied to the output file (ADXNSxxF.DAT) in subdirectory ADX_SDT1. The xx represents the node ID. If the output file already exists, the work file is appended to the existing output file. Each command name and return code is saved and logged when the OCF logging function becomes active. After the last command in the command file has been processed, the local output file is appended to ADXNSxxF.DAT in subdirectory ADX_SDT1 on the Acting Master Controller. The local output file is deleted when the Master store controller receives a copy of the output file.

The IPL Command Processor assumes that each command in the command file is the name of a .286 file and executes it directly. Therefore, you cannot specify shell commands, such as ERASE, directly in the

command file. To execute shell commands from the command file, you must specify command.286 with the command specified as an argument. The -c option must be present between command and the argument.

For example:

```
-3 command -c erase junk.dat
```

The above command erases the file JUNK.DAT. The return code for an internal command is not logged because the shell actually executes it. The return code that is logged represents the return code from the execution of command.286. The following is a list of shell commands:

<b>ASSIGN</b>	DVRUNIT
<b>DEFINE</b>	DVRUNLK
<b>ERASE</b>	MKDIR
<b>DVRLINK</b>	RMDIR
<b>DVRLOAD</b>	SECURITY

For more information on these commands, refer to the *4690 OS: User's Guide*.

To allow Data Distribution to update the command file in place on the disk, the attributes must be a compound, Distribute Per Update. Each command in the command file may be prefaced by a node identifier that allows each store controller on a LAN (MCF Network) to be customized. Each command must be on a separate line and ended by ASCII carriage return and line feed characters.

**Note:** The Distributed File Utilities may not distribute ADXILIOF.DAT as expected.

There are three options that may precede the command. Each option should begin with the switch character (-). The options are:

1. The Exit On Error option: -x
2. A time frame indicator: -1, -2, or -3. The default indicator is -3.
3. A node indicator: -nXX. XX is the two-character node ID. The default is the local node ID.

**Note:** Each command must be entered on one line. The commands must not be split between lines.

---

## Example of a Command File

In the example that follows, the first four commands are executed on controller DD only. USERAP1 and USERAP2 are executed on all store controllers.

```
-nDD -1 chkdisk c:
-nDD -1 adxcsw01 c: -f
-nDD -2 rename c:\adx_spgm\adxrt1sl.286 c:\adx_sdt1\adxrt1sl.bad
-nDD -2 copy c:\adx_sdt1\adxxxxx1.286 c:\adx_spgm\adxrt1sl.286
-3 c:\adx_upgm\userap1
-3 c:\adx_upgm\userap2
```

**Note:** If a command fails to terminate, you can abruptly end the command and the IPL Command Processor by pressing F1 while the command is displayed. The IPL completes. However, when you press **F1**, no other IPL Command Processor commands are processed during the IPL. The message recorded in the System Log indicates that you ended the command by pressing F1.

---

## Chapter 13. Using the Staged IPL Utility

---

### Requirements

This utility is useful only if you enable the Multiple Controller Feature, configure the controllers to allow backup, and have previously completed an Active Software Maintenance. By only IPLing one controller at a time, the terminals automatically switch to an active controller. On most systems, the order can be chosen so that the terminals end up on their primary controller. Automatic resume may be necessary in some topographies, especially those that have two TCC adapters on different controllers that back each other up. See “Recommended Use” on page 243 for more information.

---

### Capabilities

This section describes the capabilities of the Staged IPL Utility.

### TCC Network Considerations

The main benefit of the Staged IPL Utility is that at least one controller is available at all times during the Activate Software Maintenance upgrade of an application to run the terminals, provided that store controller backup is allowed. During utility execution, the terminals can switch controllers up to four times; each switch can cause an interruption of checkout lasting 6 to 10 seconds.

### Applications Only

The Staged IPL utility can cause a dump IPL code loop if used for activating either the operating system or LAN (MCF Network) because there is a brief time when both controllers are up and running different software levels of code. For this reason, the Staged IPL utility does not permit activation of either the operating system or LAN with the staged IPL.

### Incompatible Software Levels

The same exposure of incompatible software levels exists for applications. You must test each application maintenance package on a test system using the Staged IPL Utility with a long wait time to assess the compatibility of the software levels. If there are changes in the file formats or message formats, you must consider what effect these would have during the brief time that both controllers are up and running different software levels of code.

If your test reveals that an incompatibility problem could result from running the LAN (MCF Network) at different software levels on each controller, you can minimize this exposure by taking the following precautions:

- Make the master store controller and file server the last node to IPL. Thus, the files that are open by the terminals continue to use the older level files until the terminals are ready to load. This arrangement prevents an incompatibility problem when terminal code has a corequisite change to controller code or data files.
- IPL the terminals immediately after they come on line with a controller that has activated the new maintenance, but only if an IPL of the terminals is necessary to load new terminal code. Use the TL parameter on the ADXCSTOL command, not the disable terminal IPL function of the ADXSERVE command. If the terminals must be loaded, the TL parameter is recommended under any circumstances. If terminal code has a corequisite change to controller code or data files, avoid the disable terminal IPL function of the ADXSERVE command so that old terminal code does not try to interface with new controller code. If there is no corequisite change with controller code or data files, the disable terminal IPL function is recommended while cashiers are signed on. (See “Loading Terminal Storage” on page 246.)
- Minimize the wait time between controller IPLs so that the controllers do not have much time to communicate with each other. This prevents an incompatibility problem when controller code has a corequisite change to code or data files on another controller. However, this option is undesirable

because of the time required for all of the background applications and runtimes to load. It may be necessary to set the wait time to zero seconds if the software levels are found to be incompatible. If you do so, any terminal request to the controller will take more time than usual because the request must compete with the loading of background applications and runtimes.

**Note:** These steps are unnecessary for most maintenance; they are only precautions you should follow when the software might be incompatible. Even then, it may not be necessary to follow all these suggestions. Use those that provide the most safety with the least inconvenience.

---

## Application Interface

This section describes the application interfaces that you can use with the Staged IPL Utility.

### Apply Software Maintenance

Before running the Apply Software Maintenance (ASM) Utility, run ADXCST0L with the NI RCP parameter to build the activation file, ADX\_SPGM:ADXCSTAF.DAT. This file instructs the IPL code to activate maintenance upon the next IPL, but this procedure does not IPL the controllers. You can then IPL the controllers manually or use this utility to IPL each controller individually.

A dump or a power outage after you select NI and before the Staged IPL Utility runs can cause ASM to run prematurely during the IPL. That should not cause any problem except for a longer than normal IPL, which would cause terminals to be offline for a longer than normal period. You can minimize this window by running the Staged IPL Utility immediately after running ADXCST0L with the NI parameter.

The activation file created by running ADXCST0L with the “NI” parameter will always be erased by the Staged IPL Utility or by the IPL portion of Apply Software Maintenance. If the Staged IPL Utility does not complete successfully and must be restarted, you must also restart ADXCST0L.

### RCP Command

The Staged IPL Utility is an RCP command named ADXCS50L, which is placed in the RCP command file, usually immediately following the ADXCST0L command.

#### Format:

`ADXCS50L i w t n n...`

where:

- i** = The RCP status file reset indicator:
  - Y** = Reset RCP status file before the command executes.
  - N** = Do not reset RCP status file.

**Note:** Normally, this indicator is N because it usually runs after ADXCST0L. Also, you would not want to erase resulting messages from ADXCST0L.

- w** = Wait time

This indicator is the number of seconds to wait after the remote node comes up. The node is considered up by this utility after the logo is first displayed on the controller, and the background applications have begun to load and start. The wait time should usually be the estimated amount of time it takes to load and start the background applications. The value must be between 0 and 2147483 (about 24 days). The value must not have a decimal point, comma, or minus sign.

- t** = Timeout value



This indicator is the number of minutes to wait for the remote node to come up. If the node does not come up within the number of minutes specified, this utility ends with an error message and attempts to cancel maintenance on the controllers that have already successfully activated maintenance. The value must be between 10 and 35791 (about 24 days). The value must not have a decimal point, comma, or minus sign.

**n** = Node ID

The node IDs are sets of two-character designations for the controllers to be IPLed. The controllers are IPLed in the order of the node IDs specified with this command. If the controller on which this utility runs is on the list, it must be last. Otherwise, the IPL ends this utility and the subsequent nodes are not be IPLed.

### Examples:

```
ADXCS50L N 300 120 DD CC
```

This example does not reset the RCP status file. It requests that controller DD be IPLed first. After controller DD comes up, wait 300 seconds (5 minutes) for the background applications to finish loading and then IPL controller CC. If DD does not come up within 120 minutes, do not IPL CC.

```
DD::ADXCS50L N 0 180 CC DD
```

This example does not reset the RCP status file. It requests that this utility be run on controller DD so that you can IPL controller CC first. After CC comes up, immediately IPL controller DD. If CC does not come up within 180 minutes, do not IPL DD.

## Error Recovery

If any error is detected while IPLing the controllers, this utility does not IPL the remaining controllers. If a controller has already IPLed before the error is detected, this utility attempts to cancel the maintenance from all controllers that have successfully activated maintenance. To cancel maintenance, the activation must have been in test mode, and the controller must have come back up. If the error was a timeout waiting for a controller to IPL, the maintenance of that controller cannot be canceled, leaving the controllers at different software levels.

Some examples of errors that can stop the IPL sequence after controllers have already started IPLing are:

- A timeout waiting for the controller to complete the IPL. The timeout value is specified as a parameter.
- An error while activating maintenance during the IPL. The return code of the activation process is kept in a temporary file, ADX\_SDT1:ADXCS5TF.DAT. This error is rare and would be accompanied by a W638 message in the System Event Log. If a W638 is logged, the activation of maintenance has failed and remains in the same state as before the attempt to activate maintenance. The new maintenance might be in the ADX?SMNT? subdirectory, depending on when the error occurred.
- The request to IPL could not be communicated to a remote controller for a reason other than not being active on the LAN (MCF Network).
- One of the controllers that has come up with the new software has dumped.

All progress and error messages are logged in the RCP status file on the acting master controller. If messages cannot be written to the acting master, which happens if it is not the last controller to IPL, messages are written to the local controller. Before this utility terminates, the messages are transferred to the acting master. If this transfer fails, the messages must be gathered from both the acting master and the local controller.

## Recommended Use

This section provides some recommendations for using the Staged IPL Utility.

## Test Mode

Activation of maintenance should be in test mode. This mode allows automatic cancellation of maintenance if anything goes wrong. If disk space is a concern, the maintenance can be accepted after the RCP status file has been checked to verify the successful completion of the IPLs. An acceptance of maintenance in test mode does not require an IPL of the controllers. If the RCP status file indicates that the controllers are now at different software levels, you must run an appropriate ADXCSTOL command to direct the controllers to start running the same level of software.

## Wait Time

Before using this utility in a live store, you should apply the maintenance to a test system using this utility with a very long wait time. Observe how well the system runs with the controllers at different software levels.

If the new software is highly compatible with the old software, the wait time should be about 300 seconds. This gives the applications time to load and open all of the necessary files. Some systems might take longer to load applications, depending on the number and complexity of the applications to be loaded. If the wait time expires before all of the applications are loaded, any terminal request to the controller will take longer than usual because the request must compete with the loading of background applications and runtimes.

If testing reveals an incompatibility problem with running the old and new software on another controller, the wait time should be zero.

## Timeout Value

Set the timeout value long enough so that it expires only when a controller has taken too long to come up. If the timeout does expire, this utility attempts to cancel the maintenance that has already been successfully applied to other controllers. Make sure that you allow time for exception log processing, for an IPL to activate configuration changes, and for other contingencies; 120 minutes should be adequate for most systems. Set the timeout value short enough so that all controllers are back up and operating at the same software level in time for the busiest part of the day.

**Examples:** Assume that you have a four-controller LAN requiring 100 minutes per controller to activate maintenance through an IPL, reconcile the exception log, and load the background applications. Assume that the timeout value chosen is 120 minutes and that the third controller does not come up within that time. The total elapsed time for the first two controllers to IPL is 200 minutes, plus 120 minutes for the third controller to timeout, and 200 minutes to cancel maintenance on the first two controllers. This time totals 520 minutes. Calculate the worst case situation for your system, and choose the timeout value accordingly.

## Where to Run the Staged IPL Utility

You can run the utility on any controller by putting the controller ID in front of the command. The controller running ADXCS50L must be IPLed last. Otherwise, this utility will end when the controller running it is IPLed.

### Examples:

```
DD::ADXCS50L N 0 120 CC DD
```

This command causes the Staged IPL Utility, ADXCS50L, to run on the controller DD.

```
AA::ADXCS50L N 0 120 CC DD
```

This command causes the Staged IPL Utility, ADXCS50L, to run on the acting master. If DD is not the acting master store controller, the utility ends with the message:

```
CONTROLLER DD MUST BE LAST IN THE LIST OF CONTROLLERS TO IPL
```

## IPL Order

You should IPL the primary controller before the backup controller. When the backup controller is IPLed, the primary controller will resume. Otherwise, you must provide some means for the resume operation, for example, configure automatic resume, resume manually, or IPL the backup controller.

### Example:

If CC is the primary store controller and DD is the backup store controller, the command should be:

```
DD::ADXCST0L N 0 120 CC DD
```

If CC is running one TCC Network and DD is running another, and they are backing each other up, you should activate automatic resume.

You should IPL the acting master last. Because the RCP status file is opened on the acting master, messages to be written while the acting master is down must be saved in the local RCP status file and written to the acting master RCP status file after it comes back up. This should not be a problem unless there are errors trying to open files and or moving the messages.

If there are two controllers and they back up each other, no sequence will result in both controllers returning to their primary node (backup or primary) unless the resume is automatic or manual.

## Load Terminals

If you apply maintenance that must be loaded in the terminal to make it active, you must load terminal storage in all terminals. You should use the TL parameter of ADXCST0L to accomplish this. The terminals do not reload until the TCC Networks switch to a controller that has been IPLed to apply the new maintenance. Also, you should disable loading of terminal storage whenever a cashier is signed on. See “Loading Terminal Storage” on page 246 for more information about disabling loading of terminal storage.

Other methods exist to load the terminals, but they all require a separate action that must be invoked after the controllers have come back up. You can invoke one of the following after checking the RCP status file for errors:

- Use the ADXCST0L N 11 RCP command.
- Press **SysReq** to access the STORE CONTROL FUNCTIONS menu. Then, select the **Load Terminal Storage** option.
- Power off half of the terminals at a time so that there are always some terminals online. This only works if memory retention is disabled.

## User Applications that IPL the Controller

If you have an application that runs whenever there is an IPL or that detects an IPL, ensure that the actions taken by that application do not cause the controller to IPL or otherwise interfere with the IPLing of the controllers. Do not manually or otherwise IPL any controller while this utility is running. Allow this utility to perform all IPLs until it has finished.

## Ideal System

The following summarizes the requirements presented in this chapter:

- Run ADXCST0L first with the NI RCP parameter.
- Put the controller on which this utility is running last on the list.

The following list summarizes the recommendations presented in this chapter:

- IPL the controller acting as primary before the controller configured to support backup.
- If the terminals need to be loaded, use the TL parameter of ADXCST0L.
- Use a wait time of 300 seconds.
- Use a timeout value of 120 minutes.
- IPL the acting master last.
- ADXCST0L should activate the software in test mode.

An example of an ideal system is:

```
CC is the acting master and file server.  
DD is the alternate master and alternate file server.  
CC is the backup store controller.  
DD is the primary store controller.  
Allow backup is active.  
Automatic resume can be active or inactive in this example.
```

The command file should be:

```
AA::ADXCS0L Y 1G TL NI  
AA::ADXCS50L N 300 120 DD CC
```

**Note:** If CC is not the acting master, this utility ends with the message:

```
CONTROLLER CC MUST BE LAST IN THE LIST OF CONTROLLERS TO IPL
```

---

## Loading Terminal Storage

Terminals must be loaded to make terminal application changes active. However, loading terminals while operators are running customer checkout would defeat much of the usefulness of this utility. Therefore, the following two exit steps are recommended to significantly reduce terminal downtime.

ADXSERVE (function 54) disables the loading of terminals. If this ADXSERVE call was made in the user exit for operator signon, and enabled (function 53) again in the user exit for operator signoff, the load all terminals request would only load terminals that were signed off. When the remaining terminals sign off, each then reloads. All terminals will eventually load, but not while the operator is running customer checkout. To use this recommendation, the old terminal code must be compatible with the new controller code.

## Enable Terminal IPL

This function enables terminals to reload. If terminals were to be reloaded earlier but could not because you had disabled IPL, making this call causes the terminals to reload.

**Note:** This function does not prevent a terminal from dumping or reloading as a result of a request to load a specific terminal.

For the enable terminal IPL parameters, see “Using the enable IPL function” on page 307.

## Disable Terminal IPL

This function prevents the automatic reload that can occur when a terminal comes online, and it enables the terminal application to use effectively the terminal RAM disk support for temporarily logging data. In most cases, when the controller comes back online, the terminal application transfers the logged data to the controller.

For the disable terminal IPL parameters, see “Using the disable terminal IPL function” on page 307.

---

## Messages

The following table explains the messages that can appear when you use this utility.

*Table 23. Staged IPL Utility Messages*

Message	Description
Request is only valid on a LAN (MCF Network) system.	The Multiple Controller Feature is not active.

Table 23. Staged IPL Utility Messages (continued)

Message	Description
ADXCST0L has not completed successfully.	The Apply Software Maintenance activation file, ADX_SPGM:ADXCSTAF.DAT, does not exist. Either ASM was not run beforehand, or ASM had an error that was logged in the RCP status file.
Request is not allowed while activating OS or LAN (MCF Network).	The Apply Software Maintenance activation file, ADX_SPGM:ADXCSTAF.DAT, indicates that either the Operating System or the Multiple Controller Feature was chosen to apply maintenance using the staged IPL. This situation is not allowed because there will be a brief time when the controllers are up and running at different software levels. In many cases, this would cause a communication problem between the controllers. Results would be unpredictable.
The wait time must be numeric characters.	The wait time cannot contain periods, commas, minus signs, letters, or any characters other than 0 through 9.
The wait time must not exceed 2147483 seconds.	The largest number allowed is 2147483.
The timeout value must be numeric characters.	The timeout value cannot contain periods, commas, minus signs, letters, or any characters other than 0 through 9.
The timeout value must not exceed 35791 minutes.	The largest number allowed is 35791.
Node names must have a length of two characters.	The node name lists must be of the form "CC DD". There should not be commas separating the parameters.
Controller xx must be last in the list of controllers to IPL.	The controller that is running this utility must be last in the list. To run on another controller, add the node ID to the front of the command, as in the following example:  DD::ADXCS50L N 0 120 CC DD
Controller xx is not active on the LAN (MCF Network).	The controller indicated is not able to receive commands using the Multiple Controller Feature.
Status of controller xx could not be verified.	An ADXSERVE command issued to the remote node to test the ability of the controller to communicate failed.
Controller xx could not be IPLed.	An attempt to IPL the remote node failed. Either the request returned a negative return code, or the controller remained up after acknowledging the request. If no failing return code was returned from the request, the user-specified timeout value was used to prevent a long wait for the controller to go down.
Controller xx has IPLed.	The remote controller has successfully received the command to IPL. If an error is detected, this utility will not try to IPL the remaining controllers. Use this message to determine which controllers were IPLed before the failure occurred.
Controller xx became inactive after IPL.	The remote controller IPLed, then came up to at least the point when background applications are started. After the specified wait time, a query of the node returned an error indicating that the controller is not active on the LAN (MCF Network).
Controller xx is not configured.	The remote controller has not been defined by configuration.
Controller xx did not come up within nn minutes.	The remote controller has not come up within the number of minutes you specified as the timeout parameter.
None of the remaining controllers will be IPLed.	Because of an error reported previously, the IPL sequence was ended before all controllers were IPLed.
None of the controllers have been IPLed.	Because of an error reported previously, the IPL sequence was ended before any controllers were IPLed.
Return Code is 8xxx4xxx.	See the <i>4690 OS: Messages Guide</i> for further information on interpreting the error.

Table 23. Staged IPL Utility Messages (continued)

Message	Description
ADXSERVE Return Code is -1xxx.	See “ADXSERVE (requesting an application service)” on page 293 to interpret the error.
Create of temporary work file on node xx failed.	The temporary file, ADX_SDT1:ADXCS5TF.DAT, could not be created on the remote node. This utility creates the file so that the IPL portion of ASM can put the return code in it. Use this file to verify that ASM ran properly during the IPL.
xx is not a valid node ID.	The node ID is not two uppercase letters.
Insufficient number of parameters.	You must supply at least four parameters for this function to work. See the section about the format of the command.
Controller xx cannot be listed more than once.	The node ID was the same as another node ID previously listed in the parameter list. If it is not listed twice, it is the same as another logical drive that points to the same controller.
Activation of maintenance on controller %c%c failed.	The IPL portion of ASM that moves files failed. A W638 message should be in the System Event Log. Maintenance will be canceled automatically if possible.
See the W638 message in the System Event Log.	The IPL portion of ASM that moves files failed. A W638 message should be in the System Event Log. Maintenance will be canceled automatically if possible.
Cancel failed because activation was not test mode.	At least one controller was IPLed to activate maintenance. When a problem prevented the remaining controllers from activating maintenance, the cancel attempt failed because the activation was not test mode.
The controllers are now at different maintenance levels.	Some of the controllers were successfully IPLed to activate maintenance, but a problem occurred that prohibits the remaining controllers from activating maintenance. It was not possible to cancel maintenance.
The reset parameter must be a Y or N.	The first parameter must be a Y or N.
Attempting to cancel maintenance on controller xx.	At least one controller was IPLed to activate maintenance, but a problem prevented the remaining controllers from activating maintenance. The utility now attempts to change the activation file command from test to cancel and IPL the controllers that have already activated maintenance in test mode.
Attempt to cancel maintenance on controller xx failed.	At least one controller was IPLed to activate maintenance, but when a problem prevented the remaining controllers from activating maintenance, the cancel attempt failed.
The timeout value must be at least 10 minutes.	The timeout value must be at least long enough to allow time for the controller to IPL and activate maintenance.
The failing parameter is xx.	The error message previously reported points to the indicated parameter.
Remove diskette from drive A: on node xx.	A diskette is in the A: drive of one of the controllers, and is preventing the successful activation of maintenance. Remove the diskette, and restart both ADXCST0L and ADXCS50L.

---

## Chapter 14. Using the Java Terminal Offline Function

---

### Understanding the Java Terminal Offline Function

**Attention:** The use of the Java Terminal Offline Function (JavaTOF) is required when running Java 2 applications on 4690. This is because the Java 2 JVM requires many files, such as its font files, to always be accessible.

The terminal offline condition is a serious problem for any 4690 operating system terminal that is running Java applications. In Java, when a class is used by the application, access to a hard disk is required to load the class if that class is currently not in memory. If a terminal is offline, the access request to the hard disk on the controller fails causing the Java Virtual Machine (JVM) on the terminal to possibly abend when a `java.lang.ClassNotFoundException` is thrown. There is no easy recovery from this condition, and it is nearly impossible to continue using the application until the network connection is reestablished with the controller.

The operating system offers a Java Terminal Offline Function (JavaTOF), which is a Java-based program. JavaTOF allows you to create a ZIP file containing all Java classes and a ZIP file that contains all resource files that are required by an application. The ZIP files are then loaded onto the terminal RAM disk which allows the application to access the required Java classes and resource files from the terminal RAM disk. Then, if a terminal loses its connection with the controller, the application can continue to operate.

JavaTOF is divided into three parts:

- Resource Creation allows you to extract resources from existing ZIP or JAR files into one single ZIP file that can be accessed by the application at all times. This JavaTOF-generated ZIP file must be placed on the RAM disk.
- Dependency checking determines the Java classes used by a given application, and then places these classes into one single ZIP file that can be accessed by the application at all times. This JavaTOF-generated ZIP file must be placed on the RAM disk.
- Application startup runs the application and also hides the implementation details from the application. Java applications that run with the provided JavaTOF solution directly interface with this final part.

**Note:** JavaTOF is not supported in Java 1.6 in the 4690 OS V6 Enhanced Mode.

4690 OS Version 4 improved JavaTOF configuration. Refer to the *4690 OS: User's Guide* for detailed information about the enhanced terminal preload and Java configuration functions.

---

### Requirements for Applications Using the JavaTOF Solution

To run an application in a terminal offline environment, the application must perform the following steps:

- The application needs to be modified so that all references to resource files of any kind (such as images, icons, files to be loaded) are handled appropriately. Copying these files to NVRAM, or to the local disk on the terminal, or including them as resources in the application's JAR or ZIP files are possible solutions.
- For applications using Oracle JFC Swing, all non-Metal Look and Feel Java classes, such as Motif Look and Feel, must be included in the inclusion file passed into the JavaTOF class. The JavaTOF solution provides only for Metal Look and Feel classes to be dependency checked automatically using the `depchecking.SWING.version` property.
- It is the application developer's responsibility to handle any application file I/O processing (such as reading, writing, and reconciling) that occurs in a terminal offline environment.



---

## Resource Creation

The JavaTOF solution works by providing a way to access the Java classes that are necessary to run a Java application in a terminal that is offline. However, to run correctly, these applications still need to access resources such as image files (GIF and JPEG files), help files (HTML), and other data files. Many of these resources are bundled with the ZIP or JAR file that is used to run the application.

The JavaTOF solution provides a function, referred to as resource creation, which extracts the resources from the original ZIP or JAR files and bundles them into a new ZIP file. This new JavaTOF-generated ZIP file must be copied to the terminal RAM disk. Configure the terminal classpath to point to this resource file on the RAM disk to allow the application to access the resources.

---

## Dependency Checking

The JavaTOF solution provides a function for determining the Java classes that a given application references, referred to as class dependency checking, and places these classes in a ZIP file. Given a starting class, the dependency checking function finds all the class dependencies for that class, finds the class dependencies of those Java classes, and continues until all possible Java classes that can be referenced are found. The classes found are then placed in a ZIP file for loading on the terminal.

## Dependency Checking Limitations

JavaTOF cannot perform dependency checking on classes created dynamically using the *Class.forName()* call. To avoid this limitation, the inclusion and exclusion files are created to provide hints. If an application uses the *Class.forName()* call, these classes must be listed in the inclusion file.

The exclusion file performs the opposite service. Any classes listed in this file are excluded from dependency checking. This feature is useful for applications where it is known that a certain code path will not be invoked and, therefore, can reduce the total number of classes that need to be loaded.

Graphical Java applications that use the Oracle JFC Swing components might need to provide an inclusion file. The Swing code uses the *Class.forName()* call in many instances where a determination is made about which look and feel to use for a particular Swing component. All Swing classes for the Metal Look and Feel can be automatically included for dependency checking. For other look and feel components, all Swing classes for that look and feel, for example Motif Look and Feel, should be included in the inclusion file.

However, if the JavaTOF solution is to be run on an off-the-shelf application or an application where you have no knowledge of the code available, a method can be used where all the ZIP and JAR application files can be passed in for dependency checking.

---

## Using the JavaTOF Classes

Two classes are provided for executing the JavaTOF solution:

- **TOFCreate**  
This class is responsible for resource creation and dependency checking.
- **TOFStartApp**  
This class is used to start an application on a terminal.

Usage of the two classes is as follows:

```
java com.ibm.OS4690.TOF.TOFCreate <property file>
```

and

```
java com.ibm.OS4690.TOF.TOFStartApp <property file>
```



See “JavaTOF Property File” on page 251 for more information on the JavaTOF property file.

---

## Setting up the Java Classpath for Running the JavaTOF Solution

TOFCreate runs on the controller and requires that OS4690.ZIP be on the controller classpath. TOFCreate looks for classes that are referenced by the terminal application in the controller classpath or the depchecking.use.jarfilelist specification for the application. The controller classpath specification for the application's ZIP or JAR file overrides the depchecking.use.jarfilelist specification for the application's ZIP or JAR file. It is recommended that any ZIP or JAR file containing classes, that are referenced by the terminal application, are put into the depchecking.use.jarfilelist specification for the application.

Although dependency checking and resource creation ideally place all necessary classes and resources in one or two archives, it is possible that a class or resource might be overlooked due to a *Class.forName()* or resource reference. For this reason, when running TOFStartApp, ensure that all classpath entries used to generate the JavaTOF archives remain on the classpath. It is important that the JavaTOF-generated archives be placed at the front of the classpath, as this prevents any unnecessary accesses to the controller.

### Notes:

1. Because JavaPOS makes extensive use of *Class.forName()*, put the JavaPOS archives either in the TOF JAR include list or directly in the terminal RAM disk.
2. The Terminal Classpath needs to be updated with the ZIP and JAR files, which are needed to run the Java application instead of a generic classpath statement.

---

## JavaTOF Property File

The JavaTOF solution uses a properties file for setting up parameters to run JavaTOF.

An example properties file for the Java browser is shown below. This file is divided into three sections:

- Resource creation
- Dependency checking
- Application startup

It is important to note that this example includes all of the properties that are used by the JavaTOF classes. If you only want a particular part to run and are concerned about readability, then only that section needs to be included in the property file that is created.

The property file, that is passed as an argument to TOFCreate, is placed in the generated dependencies output file. When invoking TOFStartApp, TOFStartApp looks for the property file argument within the classpath. Therefore, a single property file serves both TOFCreate and TOFStartApp.

```
#####
# This is a sample JavaTOF property file
#####
#
#
#####
# Resource Creation Properties
#####
resource.creation.run=no
resource.creation.debugon=yes
resource.creation.use.resourcefilelist=f:/TOF/test/resource.rci
resource.creation.outputfile=f:/TOF/test/resource.rco
#
#####
# Dependency Checking Properties
#####
depchecking.run=no
depchecking.debugon=yes
```

```

depchecking.application.classname=ice.elite.ELite
# -- Dependency Include and Exclude Files
depchecking.use.includefile=g:/TOF/test/include.dui
depchecking.use.excludefile=g:/TOF/test/exclude.dux
# -- Dependency Using JAR file list
# depchecking.use.jarfilelist=f:/TOF/test/jarlist.duj
depchecking.SWING.version=1.1.1
depchecking.javapos.version=1.4
# -- Dependency Output Result file
depchecking.outputfile=c:/tof/elite/elite.duo
#
#####
# Application Properties
#####
application.classname=ice.elite.ELite
application.args=

```

### Notes:

1. The JavaTOF property file name must be at least 4 characters in length.
2. When assigning a file name to one of the properties, make sure that the file separator used in the file location is one that is expected by the platform running the Java application. This information can be found in the Java file.separator property. On the 4690 platform, all file separators should use the forward slash character (/).
3. You can comment out any line of the property file by placing the pound symbol (#) as the first character in the line.

## Resource Creation

This section in the JavaTOF property file specifies how to extract the necessary resources that an application requires to run with the JavaTOF solution. The following table shows the current property values that can be used.

Property Name	Description	Valid Values	Optional/Required
resource.creation.run	Should the resource creation be run?	Yes, No	Optional (all other resource creation properties are ignored if this value is not Yes)
resource.creation.debugon	Turn debugging on?	Yes, No	Optional
resource.creation.use.resourcefilelist	The name of the file containing all the JAR or ZIP files where resources should be extracted.	Valid file with a list of JAR or ZIP files	Required (if resource.creation.run property is Yes)
resource.creation.outputfile	The name of the resource file created.	Valid file name	Required (if resource.creation.run property is Yes)

The following provides a detailed description of the resource creation properties in the JavaTOF property file.

### resource.creation.run

Determines whether resource creation should be run. Any value other than Yes causes the resource creation to not run.

### resource.creation.debugon

Determines whether debugging should be turned on. This debug information is written to STDOUT. Any value other than Yes results in no debug information being created.

### resource.creation.use.resourcefilelist

Lists a file with the location of all JAR and ZIP files where resources are to be extracted. For

example, if resources are to be extracted from two JAR files, C:/RESOURCES/FIRSTRESOURCE.JAR and C:/RESOURCES/SECONDRESOURCE.JAR, then the file, C:/RESOURCES.TXT, can be created with the following lines:

```
c:/resources/firstresource.jar  
c:/resources/secondresource.jar
```

and the property is set as:

```
resource.creation.use.resourcefilelist = c:/resources.txt
```

An error occurs if the resource creation is run and this file is not defined.

### resource.creation.outputfile

This property is where the new ZIP file containing the resources extracted should be placed. If the resource creation runs successfully and no resources exist in the resource files listed, then no new resource file is created. An error occurs if the resource creation is run and this file is not defined.

**Note:** This property can have the same value as depchecking.outputfile.

## Dependency Checking

This section in the JavaTOF property file specifies how the dependency checking part of the JavaTOF solution should be invoked. The following table lists the current property values that can be used:

Property Name	Description	Valid Values	Optional/Required
depchecking.run	Should JavaTOF dependency checking be run?	Yes, No	Optional (all other dependency checking properties are ignored if this value is not Yes).
depchecking.debugon	Turn debugging on?	Yes, No	Optional
depchecking.application.classname	The Java class where the main method is invoked for the application.	Java class	Required (if depchecking.run property is Yes)
depchecking.use.includefile	Other .CLASS files to be included in the dependency checking.	Valid file with a list of Java classes	Optional
depchecking.use.excludefile	Other .CLASS files to be excluded from the dependency checking.	Valid file with a list of Java classes	Optional
depchecking.use.jarfilelist	Other JAR or ZIP files that should have dependency checking done on.	Valid file with a list of JAR and ZIP files	Optional
depchecking.use.lastrun	Has been deprecated and is no longer honored. If the user specifies the property, then a warning is displayed and execution continues.	Not applicable	Not applicable
depchecking.SWING.version	Version of Swing that the JavaTOF class is to be used on.	1.1.1 1.0.x etc.	Optional
depchecking.javapos.version	Version of JavaPOS that the JavaTOF application will use	1.4	Optional
depchecking.outputfile	The name of the dependency file created.	Valid file name	Required (if depchecking.run property is Yes)

The following list of property names provides a detailed description of the dependency checking properties in the JavaTOF property file.

#### **depchecking.run**

Determines whether dependency checking should be run. Any value other than Yes causes the dependency checking to not run.

#### **depchecking.debugon**

Determines whether debugging should be turned on. This debug information is written to STDOUT. Any value other than Yes results in no debug information being created.

#### **depchecking.application.classname**

Shows the Java class from which to start the class dependency checking (this class should have the main method). This class gives as accurate a dependency check as can be obtained. In the example main method shown below, class dependency checking first occurs on class X:

```
public class X
{
    public static void main (String[] args)
    {
        Class A = new A();
        Class B = new B();
    }
}
```

When the dependencies of this class (and their dependencies) have been found, dependency checking then continues on class A, and finally class B. An error is thrown if the dependency checking is run and this property does not exist. The Java class should be specified using the '.' notation and without the .CLASS suffix.

#### **depchecking.use.includefile**

If the *Class.forName()* command is used within an application and it is known where this command is used, then the classes passed into this command should be listed in a file pointed to by this property. When this property is defined and the file exists, all the classes in this file are added to the list of classes that need to be dependency checked. The format for the include file is as follows where the names of the classes are listed without the .CLASS extension:

```
app.example.class1
app.example.class2
```

#### **depchecking.use.excludefile**

Dependency checking might find many unnecessary classes. If you know when certain parts of the application are not being used, you can reduce the number of classes to be preloaded to improve both the application startup time and the amount of terminal RAM disk being used. For example, if the LayAway function is not used, then an exclude file could be created with the name of the Java class handling that function. This action causes the Java class to be excluded from any dependency checking. The format for the exclude file is as follows where the names of the classes are listed without the .CLASS extension:

```
app.Layaway.Feature
app.Layaway.Discount
```

The exclude file also has limited wildcard support (the wildcard can only be used at the end of the text), so the above file could be simplified by the following line that would exclude all classes that have the app.Layaway. path notation:

```
app.Layaway.*
```

#### **depchecking.use.jarfilelist**

If the application being run is an off-the-shelf application or a third-party application and the only knowledge you have of the application is that the *Class.forName()* call might be used, then the suggested method of dependency checking is using this property. For example, if an application

uses two JAR files, C:/TEST/APP1.JAR and C:/TEST/APP2.JAR, then a file named, C:/APPJARS.TXT, could be created with the following lines where the names of the JAR and ZIP files are listed one after another:

```
c:/test/app1.jar  
c:/test/app2.jar
```

If this property then points to the C:/APPJARS.TXT file, this calls all the Java classes from each of the JAR files listed and performs dependency checking on every one of the classes.

#### **depchecking.use.lastrun**

This property has been deprecated and is no longer honored. If the user specifies the property, then a warning is displayed and execution continues.

#### **depchecking.SWING.version**

This property is used to automatically include all Metal Look and Feel Swing classes in the dependency check. For the Swing 1.1.1 version, the following classes need to be dependency checked for a Java application using the Swing Metal Look and Feel:

```
javax.swing.plaf.basic.*  
javax.swing.plaf.metal.*
```

If this property has the value of 1.1.1, all these classes are automatically included in the dependency checking.

**Note:** Because Metal is the default look and feel for the operating system, this property ensures these classes get dependency checked. For other Look and Feel components, such as Motif Look and Feel, it is necessary to add those to an include file. If the application being run does not use any Swing components, this property does not need to be defined.

#### **depchecking.javapos.version**

This property is used to automatically include all JavaPOS classes in the dependency check.

#### **depchecking.outputfile**

This property is where the ZIP file, with the Java classes that this application depends on, is written. An error occurs if dependency checking is run and this file is not defined.

**Note:** This property can have the same value as resource.creation.outputfile.

## **Application Properties**

This section in the JavaTOF property file provides specific directions on how the application properties part of the JavaTOF solution should be invoked. The following table shows the current property values that can be used.

Property Name	Description	Valid Values	Optional/Required
application.classname	Java class that contains the main method to be invoked.	Java class	Required (if running TOFStartApp)
application.args	Arguments that are used to run the Java application.	All arguments to be passed in should be listed on this property line.	Optional (if no value is defined, then no arguments are assumed)

The following provides a detailed description of the application properties in the JavaTOF property file:

#### **application.classname**

Specifies the Java class that contains the main method to start the application. An error is thrown if TOFStartApp is run and this property does not exist. The Java class should be specified using the '.' notation and without the .CLASS extension.

### **application.args**

The arguments that would normally be passed in to start the Java application are given by the application.classname property. If the application takes no arguments, then this property does not need to be defined.

## **The JavaTOF Property File Editor**

The operating system provides a JavaTOF property file editor to simplify the task of editing and organizing JavaTOF property files. The JavaTOF property file editor is a Java utility that allows you to create, edit, and delete JavaTOF property files and associated files. It also allows you to run the TOFCreate class using those files. The editor forces a naming convention for the input and output files needed for the JavaTOF property file. When you assign a name to the property file, the JavaTOF property file editor uses the following file extensions on the file:

File Name	Description
.PRO	The actual property file
.RCI	The resource creation input file
.RCO	The resource creation file
.DUI	The dependency include file
.DUO	The dependency output file
.DUX	The dependency exclude file
.DUJ	The dependency JAR list file

**Note:** Resource Creation and Dependency Checking can be run from the GUI. The RUN button causes the TOFCreate class to run which, in turn, performs resource creation or dependency checking or both based on what is in the property file.

All files reside in the JavaTOF subdirectory.

To invoke the property file editor, enter:

```
java com.ibm.OS4690.TOF.TOFEdit xxxx
```

where, xxxx is an optional parameter to indicate where to find the JavaTOF subdirectory. The default location is C:\TOF.

The editor adheres to strict guidelines regarding the format of the JavaTOF property file. The first line within the JavaTOF property file must be a comment line containing the description. Blank lines are not allowed within the property file. Also, headers are four lines of comments. Each section within the property file has a header, and there is a header at the beginning of the file. Unused properties are commented out and ignored; the editor does not remove them from the property file.

### **Property File Editor Screens**

The JavaTOF property file editor consists of several panels that allow you to work with the different areas of the JavaTOF property file.

The main edit panel lists the property files along with their descriptions. This list includes every property file ending in .pro or .PRO found in the JavaTOF subdirectory. A message displays that indicates how many valid and incorrect property files are found. Incorrect files are files that do not follow the format expected by the editor.

From the initial panel, you can edit, rename, or delete property files.

By working through the various panels, you define which parts of the JavaTOF property file you want to run. To save any changes to the panels, press **Save**. Press **Return** to discard all changes

## Property File Editor Output

TOFEdit redirects STDOUT and STDERR to TOFEDIT.OUT and TOFEDIT.ERR. The JavaTOF property file editor does not write to these files, but Java exceptions are written to the files.

## Property File Editor Error Codes

If the JavaTOF Property File Editor cannot read or write a property file, it displays an error code. Table 24 shows the list of error codes and their explanations.

*Table 24. JavaTOF Property File Editor Error Codes*

Error Code	Description
1	.pro file could not be opened
2	The description comment is missing
3	The profile's four-line comment block is missing
4	Resource.creation.run property is missing
5	Resource.creation.run must be Yes or No
6	Resource.creation.only property is missing
7	Resource.creation.only must be Yes or No
8	Resource.creation.debugon property is missing
9	Resource.creation.debugon must be Yes or No
10	Resource.creation.use.resourcefilelist property is missing
11	Resource.creation.use.resourcefilelist has no file name
12	Resource.creation.outputfile resource is missing
13	Resource.creation.outputfile has no file name
14	Four-line comment block above Dependency Checking properties is missing
15	Four-line comment block above Resource Creation properties is missing
16	Depchecking.run property is missing
17	Depchecking.run must be Yes or No
18	Depchecking.only property is missing
19	Depchecking.only must be Yes or No
20	Depchecking.debugon property is missing
21	Depchecking.debugon must be Yes or No
22	Depchecking.application.classname property is missing
23	Depchecking.application.classname has no class name
24	Depchecking.use.includefile property is missing
25	Depchecking.use.includefile has no file name
26	Depchecking.use.excludefile property is missing
27	Depchecking.use.excludefile has no file name
28	Depchecking.use.jarfilelist property is missing
29	Depchecking.use.jarfilelist has no file name
32	Depchecking.SWING.version property is missing
33	Depchecking.SWING.version has no version number
34	Depchecking.outputfile property file is missing

Table 24. JavaTOF Property File Editor Error Codes (continued)

Error Code	Description
35	Depchecking.outputfile has no file name
36	Four-line comment block above Memory Class Loader properties is missing
51	Four-line comment block above Application properties is missing
54	Application.classname has no class name
55	Application.classname property is missing
56	Application.args must have a value or be commented out
57	Application.args property is missing
58-64	Error writing property file to disk
65	Error reading the resource.creation.resourcefilelist file
66	Error reading depchecking.use.includefile file
67	Error reading depchecking.use.excludefile file
68	Error reading depchecking.use.jarfilelist file
69	Error renaming the .temp property file to .pro
70	Error writing the resource.creation.resourcefilelist file
71	Error writing the depchecking.use.includefile file
72	Error writing the depchecking.use.excludefile file
73	Error writing the depchecking.use.jarfilelist file
74	.temp file missing while copying .tmp to .pro
75	Could not erase the old .pro file
77	Delete function could not delete the .pro file
78	Delete function could not delete the resource.creation.use.resourcefilelist file
79	Delete function could not delete the resource.creation.outputfile file
80	Delete function could not delete the depchecking.use.includefile file
81	Delete function could not delete the depchecking.use.excludefile file
82	Delete function could not delete the depchecking.use.jarfilelist file
83	Delete function could not delete the depchecking.outputfile file

**Note:** The "missing" in the error code might also indicate misplaced. An extra blank line or comment line causes a property to be "missing" from the line it should be on.

## Procedure Overview

Perform the following steps to use the JavaTOF function:

1. Make a JavaTOF directory.
2. Create the properties files and associated files.
3. Run TOFCreate against the property file to generate ZIP files containing resources and class dependencies for the terminal application.
4. List the ZIP files generated in step 3 in the Terminal RAM Disk Preload Utility's input file (named ADX\_IDT1/ADXTRMXF.DAT for the X: RAM disk or named ADX\_IDT1/ADXTRMYF.DAT for the Y: RAM disk). Then run the Terminal RAM Disk Preload Utility (ADXTRM0L -x) to create the RAM disk preload ZIP file.
5. Update the size of each terminal's RAM disk so the RAM disk preload file fits on the terminal's RAM disk.



6. Add the ZIP files to the beginning of the terminal classpath on the terminal RAM disk. The terminal load shrink is automatically rebuilt after activating the configuration, such as the terminal, system or controller, and it contains the RAM disk preload file.
7. Run TOFStartApp on the terminals with the property file that is used with the TOFCreat class.
8. If the Java terminal application running under JavaTOF is modified or whenever updates to OS 4690 occur, such as maintenance packages or new releases are received, TOFCreat should be rerun to pick up any changed classes or resources. Then, rerun the Terminal RAM Disk Preload Utility (ADXTRM0L -x) and the Terminal Load Shrink Utility (ADXRTCCL). Otherwise, after steps 1-6 have been completed for an application, it is only necessary to run TOFStartApp to run the application offline.

**Note:** When TOFCreat places the property file in the dependencies ZIP file, the name of the property file does not maintain any directory information. For example, if TOFCreat is run as follows:

```
java com.ibm.OS4690.TOF.TOFCreat c:\dir1\dir2\sample.pro
```

then, the TOFStartApp is called as follows:

```
java com.ibm.OS4690.TOF.TOFStartApp sample.pro
```

## Creating the Resource File

To create the resource file, perform the following steps. These steps can also be accomplished by using the JavaTOF Property File Editor.

1. Set the following properties in your JavaTOF property file to run resource creation:
  - resource.creation.run=yes
  - resource.debugon=yes (if a view of the resources being extracted is wanted)
  - resource.creation.use.resourcefilelist=input file name with the ZIP or JAR files that the application uses to run
  - resource.creation.use.outputfile=output file name of the new resource file to create
2. Create the input file for resource creation. For example, to create a resource file for the browser application that uses the ICEELITE.JAR file, the input file would contain the following line:

```
c:/elite/iceelite.jar
```

Any ZIP and JAR files that the application uses should be included in this list so that the complete list of resources can be captured in the new ZIP file that is created.

3. If a java.util.zip.ZipException occurs, check whether any of the ZIP or JAR files listed in the file specified by the resource.creation.use.resourcefilelist are corrupt. If there are no corrupt files, this exception has most likely been thrown due to one of two reasons:
  - If none of the files that were listed had any resources that could be extracted, then there are no resources in the listed files that the application needs to use while running.
  - If the files are not corrupt then there are no resources and Resource Creation should be turned off.

## Creating the Dependency Checking Output File

To create the dependency checking output file, perform the following steps. These steps can also be accomplished by using the JavaTOF Property File Editor.

1. To run dependency checking, set the following properties in your JavaTOF property file:
  - Depchecking.run=yes
  - Depchecking.debugon=yes (if detailed progression messages are wanted)
  - Depchecking.application.classname=your application class name
  - Depchecking.SWING.version=1.1.1 (if the application uses SWING)
  - Depchecking.javapos.version=1.4 (if the application uses JavaPOS)
  - Depchecking.outputfile=output file name of the file to contain the classes required by this application

If you know that there are no *Class.forName()* calls by the application, then it is recommended to run the dependency checking with just these properties defined. This step returns the most accurate list of Java classes used by the application and the order in which they are invoked.

If, however, the application performs the *Class.forName()* call or you do not know if this call is performed, then one of the following steps should be performed:

- If you know all of the Java classes that the application performs the *Class.forName()* call on, then the following property can be defined:
    - `depchecking.use.includefile`=input file name (with a list of all classes that get invoked using the *Class.forName()* call)
  - If there is no way to determine whether the *Class.forName()* call is used by the application, then the following property should be defined:
    - `depchecking.use.jarfilelist`=input file (with all the JAR and ZIP files that the application requires to run)
2. Run the `TOFCreate` class as follows (where `SAMPLE.PRO` is the JavaTOF property file name in this example). This step can be run on the controller or the terminal.

```
java com.ibm.OS4690.TOF.TOFCreate c:/sample.pro
```
  3. If a `java.lang.ClassNotFoundException` occurs, determine why the class is not found. Some possible courses of action to correct a missing class include:
    - Add the missing ZIP or JAR file to the classpath.
    - Classes might appear that are neither used nor packaged with the application ZIP or JAR files. If this occurs, it indicates that the missing class is not required for offline operation and can be added to the file specified by the `depchecking.use.excludefile` property.
    - If using the `depchecking.use.jarfilelist` property, add the JAR file that contains the missing class to the file specified by the property.
  4. If a `com.ibm.jikes.bmtk.ClassFileException` occurs, dependency checking was attempted on a non-Java class. If the `depchecking.use.includefile` property is defined, check to ensure that all the names in the file pointed to by that property are Java classes.

## Running the Application

To run the application, perform the following steps. The first step can also be accomplished by using the JavaTOF Property File Editor.

1. To run the application, set the following properties in your JavaTOF property file:
  - `application.classname`=your application classname
  - `application.args`=your application args
2. Run `TOFStartApp` as follows (where `SAMPLE.PRO` is the JavaTOF property file name in this example).

```
java com.ibm.OS4690.TOF.TOFStartApp sample.pro
```

If a `java.lang.IllegalAccessException` occurs, check whether the main method of the application class is a public method.

---

## Font Preloading

The Java 2 JVM accesses font files throughout its lifetime and makes the assumption that the font files are always available. The font renderer becomes highly unstable if it cannot access the font files. If the font renderer cannot access the font files, it can result in the JVM abending. For this reason, it is necessary for `TOFCreate` to place all font files on the terminal's RAM disk. `TOFCreate` looks at the appropriate `font.properties` file and determines which font files will potentially be accessed by the JVM. `TOFCreate` then renames the font files to an 8.3 short file name format appropriate for the RAM disk. These 8.3 files are automatically added to the RAM disk preload file. Note that the default behavior of the font preload should be adequate for most users. For additional information, refer to "JavaTOF Properties Files" on page 264.

Following are the font preload properties:

**fontPreload.path**

Allows a user to override the path used by TOFCreat to search for font files to be placed in the RAM disk. The value should be a list of one or more directories on the controller delimited by semi-colons, for example, `m:/fonts;m:/java2/jre/lib/fonts`.

**fontPreload.properties**

Allows a user to override the font properties file used by TOFCreat. The value of this property is the full pathname of the file, such as `m:/java2/jre/lib/font.properties.ja`.

**fontPreload.RAMdrive**

Allows a user to specify which RAM drive the fonts should be stored on. By default the RAM drive value is X:. Valid values for this property are X: and Y:.

---

## Using JavaTOF with Java 2 (Single Application)

The procedure for using JavaTOF differs between Java 1 and Java 2 due to differences in how the classpath is handled by Java 1 and Java 2. Unlike Java 1, Java 2 has two separate classpaths: the boot classpath and the user classpath.

- The boot classpath is set by default to include the appropriate JAR files in the `m:/java2/jre/lib` directory. For Java 1.4.2, this list includes the files `charsets.jar`, `core.jar`, `graphics.jar`, `security.jar`, `server.jar`, `xml.jar`, and all `ibm*.jar` files (`ibmcertpathprovider.jar`, `ibmjcefw.jar`, `ibmjgssprovider.jar`, `ibmjssefips.jar`, `ibmjsseprovider.jar`, `ibmorb.jar`, `ibmorbapi.jar`, and `ibmpkcs.jar`) in the `jre/lib` directory.
- The user classpath contains ZIP archives and individual class files as specified by the user.

The Java 2 JVM constructs its complete classpath by appending the user classpath to the boot classpath. For JavaTOF to work correctly with Java 2, the boot classpath must be defined to point to archives placed on the terminal RAM disk. This is accomplished by using the flag `-Xbootclasspath:<path>` when invoking the JVM on the terminal.

In Java 2, the Java property, `java.ext.dirs`, points to a list of directories that Java 2 searches for "JVM extensions". This directory is where such items as Input Method Extension (IME) classes are loaded. Normally, the value of this property is `l:/java2/jre/lib/ext` for a terminal, which means that the JVM could attempt to access JAR files, or other data, in that directory in an offline situation. Because the classes (JAR files) stored in the extension directory typically are not used directly by an application and, in general, are used rarely, JavaTOF does not attempt to package their contents into an archive. If you want to use any of the extension.JAR files, you need to manually add them to the RAM disk preload file and set the `java.ext.dirs` property to point to the location of the files. If you do not plan on using any extensions at all, the property can be defined to be the empty string, `-Djava.ext.dirs=`.

Currently, only IME-related files and classes are shipped with Java on OS4690. The JVM scans the JAR files during initialization to determine what extensions are installed. Later, when the first graphical Window is shown, it loads in enough of the IME classes to determine their names for display to the user. Typically, after this point, the JVM does not need to access the file any longer unless the user activates one of the input methods.

## Boot Archives

JavaTOF for Java 1 creates two archives and JavaTOF for Java 2 creates four archives. The two archives created by JavaTOF for Java 1 are a resource archive and a class dependencies archive. The four archives created by JavaTOF for Java 2 are a boot resources archive, a user resources archive, a boot class dependencies archive, and a user class dependencies archive.

The resources archives for Java 2 are specified via the *property resource.creation.outputfile*. The user resources archives for Java 2 are specified in the JavaTOF properties in the same manner as for Java 1. The user class dependencies archive for Java 2 is specified via the *property depchecking.outputfile*.

JavaTOF for Java 2 automatically generates two new boot archives with names based on the user archive names. The rules for the naming of the boot archives are:

- If the name of the user archive has a base name that is 4 characters or less (excluding the path and extension information), then the 4-character boot suffix (BDEP for dependency archives or BRES for resource archives) is appended to the base portion of the file name.
- If the name of the archive exceeds 4 characters, then the boot suffix overlays the end of the base portion to create a base portion of 8 characters in length.

The following items are examples:

- A resource archive name of ABC.ZIP yields a boot resource archive name of ABCBRES.ZIP.
- A dependency archive name of ABC.ZIP yields a boot dependency archive name of ABCBDEP.ZIP.
- A resource archive name of ABCDE.ZIP yields a boot resource archive name of ABCDBRES.ZIP.
- A dependency archive name of ABCDE.ZIP yields a boot dependency archive name of ABCDBDEP.ZIP.

**Note:** JavaTOF writes messages to the screen that state what the boot archives have been named.

## Procedure for Using JavaTOF with Java 2

The following steps should be followed to use JavaTOF with Java 2.

1. Set up the JavaTOF properties file as explained in "JavaTOF Properties Files" on page 264.
2. Execute TOFCreat at the command line using the properties file setup from step 1.

An example command line is:

```
java2bin:java com.ibm.OS4690.TOF.TOFCreat tf.prp
```

3. Add the user and boot archives to the RAM disk preload file. In the following example, the X:\ drive is being used, therefore the entries are added to the ADX\_IDT1:ADXTRMXF.DAT file.

This is an example of the lines that might be added to the RAM disk preload file:

```
C:/TOFfiles/abcBDEP.dep !Boot dependencies archive
C:/TOFfiles/abcBRES.res !Boot resources archive
C:/TOFfiles/abc.dep !User dependencies archive
C:/TOFfiles/abc.res !User resources archive
```

**Note:** In releases prior to 4690 OS V4R1, the "-c" flag was required on RAM disk preload entries when compressed files were being preloaded to the RAM disk. The flag prevented the files from being compressed again when stored in the loadshrink file. This flag is still supported in 4690 OS V4R1, but is not necessary. The form of file compression used by loadshrink in 4690 OS Version 4 Release 1 and later does not cause compressed files to grow in size the way that older compression algorithms did. ZIP files will typically be compressed slightly when stored in a loadshrink file because the file names embedded in the ZIP files will be compressed. This is especially true for TOF archives, because they contain a large number of entries.

**Note:** Entries for Java 2 font files are automatically added to the preload file by TOFCreat. These entries have the comment *##JavaTOF-Java2-TTF* (machine-generated).

4. Modify the "Class and Parameters" section of the Terminal Load Definition to override the boot classpath. Note that a TOFStartApp wrapper class with no package information is provided so there is enough room to specify the -Xbootclasspath flag. An example command line is:

```
-Xbootclasspath:x:/abcBDEP.dep;x:/abcBRES.res TOFStartApp tf.prp
```

**Note:** The class and parameters are limited to 64 characters; therefore, the archive names and property name must be very short. The above example is 64 characters long. If the parameter string is too long to fit in the available space, you might be able to rename the dependency files and property files to make the names shorter. Another method to specify a long parameter

string is to use response files. For additional information on using response files, see “Using response files to start Java programs on a terminal” on page 456.

5. Update the terminal RAM disk preload image by executing the command, `ADXTRMOL`, at the command prompt.
6. Update the terminal load-shrink file by executing the command, `ADXRTCCL`.

**Note:** It is recommended that a .BAT file be created to perform all these steps (minus the class and parameters step). The .BAT file eliminates the risk of omitting a step.

If you want to use a particular IME with your application in JavaTOF mode, the classes that make up the IME should be added to the JavaTOF archive. This is done by using the `depchecking.use.jarfilelist` property. If you want to use the Japanese IME, you must also make the dictionary file available on the terminal. This file is named `ibmbaseu.dct` and is located in the `m:/java2/jre/lib/ext` directory. The `ibmbaseu.dct` file should be added to the RAM disk preload file list so that it is loaded to the terminal RAM disk. In addition, the drive letter of the RAM disk should be added to the `java.ext.dirs` property so that the IME code can locate the file.

---

## Using JavaTOF with Multi-App

When using JavaTOF with multi-app, the procedure is slightly different so that space on the terminal RAM disk is conserved. Many classes and resources used by applications are the same. Specifically, the classes and resources used by the JVM itself are the same for each application, so there is no need to place a separate copy of the JVM archives for each application on the terminal RAM disk. The procedure differs slightly between Java 1 and Java 2.

### Notes:

1. Any system properties or JVM flags (such as, `-Xms10M`) for the primary JVM must be specified on the command line (not in the properties file). This restriction includes the setting of `java.ext.dirs`. For additional details, see “Using JavaTOF with Java 2 (Single Application)” on page 261.
2. Multi-App is not supported in Java 1.6 in the 4690 OS V6 Enhanced Mode.

## Java 2 Terminal Classpath Setup

The user classpath for the terminal should be set up to include the user archives specified in the JavaTOF properties file.

The boot classpath should be set up to include the appropriate JAR files in the `m:/java2/jre/lib` directory. For Java 1.4.2, this list includes the files `charsets.jar`, `core.jar`, `graphics.jar`, `security.jar`, `server.jar`, `xml.jar`, and all `ibm*.jar` files (`ibmcertpathprovider.jar`, `ibmjcefw.jar`, `ibmjgssprovider.jar`, `ibmjssefips.jar`, `ibmjsseprovider.jar`, `ibmorb.jar`, `ibmorbapi.jar`, and `ibmpkcs.jar`) in the `jre/lib` directory.

For example, if the JavaTOF properties file has these properties:

```
resource.creation.outputfile=c:/TOFfiles/app0.res
depchecking.outputfile=c:/TOFfiles/app0.dep
```

then this is how the classpath for the application should be configured:

```
x:/app0.res;x:/app0.dep
```

The boot classpath should be configured to contain all boot class JAR files preloaded to the RAM disk. Assuming the JAR files are preloaded to RAM disk X as indicated by the response file in “Terminal RAM Disk Preload (Java 2)” on page 265, this would be the JVM option used to configure the boot classpath:

```
-Xbootclasspath:x:/charsets.jar;x:/core.jar;x:/graphics.jar;x:/security.jar;
x:/server.jar;x:/xml.jar;x:/ibmjssef.jar;x:/ibmcertp.jar;x:/ibmjcefw.jar;
x:/ibmjgssp.jar;x:/ibmjssep.jar;x:/ibmpkcs.jar
```

The way the classpath and bootclass are specified depends on whether the application is the parent application or a child application.

For the parent application, the classpath is specified using the graphical classpath configuration utility in system configuration. The boot classpath is specified in the "Class and Parameters" field of the Terminal Load Definition using the `-Xbootclasspath:` flag. The list of boot classpath JAR files for Java 1.4.2 causes the Java parameter list to exceed the 64-character limit allowed by the terminal configuration. Because of this, you need to use a response file to specify the parameters for the Java terminal application. For additional information, see "Using response files to start Java programs on a terminal" on page 456.

For a child application, the classpath and boot classpath are both specified in the *appN.systemArgs* property. For example, the *systemArgs* property for a child application in the multi-apps properties file would look like this:

```
appN.systemArgs=<bootclasspath> -classpath x:/app0.res;x:/app0.dep
```

The text `<bootclasspath>` should be replaced by the JVM option to set the boot classpath as described above.

## JavaTOF Properties Files

A separate JavaTOF properties file should be created for each application that runs. The settings of the properties *application.classname* and *application.args* depend on whether the application is being configured as the parent application or a child application.

For a parent application, *application.classname* should be *MultiStart* and *application.args* should be the name of the multi-app properties file.

For example, if the name of the multi-app properties file is `c:/multi.prp`, then the JavaTOF properties file would have the following properties:

```
application.classname=MultiStart
application.args=R:C:/multi.prp
```

For a child application, the properties *application.classname* and *application.args* should specify the actual application that is being run.

For example, if the name of the application is `com.xyz.ChildApplication` and the arguments for the application are *arg0 arg1 arg2*, then the JavaTOF properties file would have the following properties:

```
application.classname=com.xyz.ChildApplication
application.args=arg0 arg1 arg2
```

## Multi-App Properties File

As the JavaTOF properties file, the configuration of the multi-app properties file depends on whether the application is a parent application or a child application.

**Note:** Any system properties or JVM flags (such as, `-Xms10M`) for the primary JVM must be specified on the command line (not in the properties file). This restriction includes the setting of `java.ext.dirs`. For additional details, see "Using JavaTOF with Java 2 (Single Application)" on page 261.

For the parent application, the actual application information should be provided. For example, if the name of the parent application is `com.xyz.ParentApplication` and the arguments for the parent application are *arg0 arg1 arg2*, then the multi-app properties file would have the following properties:

```
app0.classname=com.xyz.ParentApplication
app0.appArgs=arg0 arg1 arg2
```

For the child applications, the information in the multi-app properties file should specify *TOFStartApp*. The properties file passed to *TOFStartApp* should be the name of the properties file used with *TOFCreate* for



the specific application. For example, if the name of the properties file used with TOFCreate for the child application is tofApp1.prp, then the multi-app properties file would have the following properties:

```
app1.classname=TOFStartApp
app1.appArgs=tofApp1.prp
```

## Java 2 Class and Parameters

This is how the "Class and Parameters" field of the Terminal Load Definition should be configured:

```
<bootclasspath> TOFStartApp <parent application TOF properties file>
```

The text <bootclasspath> should be replaced by the JVM option to set the boot classpath as described in "Java 2 Terminal Classpath Setup" on page 263.

## Terminal RAM Disk Preload (Java 2)

The RAM disk preload list file must be configured to contain the archives listed on the boot classpath as well as all of the user archives generated by JavaTOF.

The preload list file for Java 2 would have the following entries:

```
m:/java2/jre/lib/charsets.jar
m:/java2/jre/lib/core.jar
m:/java2/jre/lib/graphics.jar
m:/java2/jre/lib/security.jar
m:/java2/jre/lib/server.jar
m:/java2/jre/lib/xml.jar
m:/java2/jre/lib/ibmjssefips.jar          -FN:ibmjssef.jar
m:/java2/jre/lib/ibmcertpathprovider.jar -FN:ibmcertp.jar
m:/java2/jre/lib/ibmjcefw.jar           -FN:ibmjcefw.jar
m:/java2/jre/lib/ibmjgssprovider.jar     -FN:ibmjgssp.jar
m:/java2/jre/lib/ibmjsseprovider.jar     -FN:ibmjssep.jar
m:/java2/jre/lib/ibmpkcs.jar            -FN:ibmpkcs.jar
c:/TOFfiles/app0.res                    ! Application 0 resources archive
c:/TOFfiles/app0.dep                    ! Application 0 dependencies archive
c:/TOFfiles/app1.res                    ! Application 1 resources archive
c:/TOFfiles/app1.dep                    ! Application 1 dependencies archive
c:/TOFfiles/app2.res                    ! Application 2 resources archive
c:/TOFfiles/app2.dep                    ! Application 2 dependencies archive
c:/TOFfiles/app3.res                    ! Application 3 resources archive
c:/TOFfiles/app3.dep                    ! Application 3 dependencies archive
```

**Note:** The first JAR files in the list above are the Java 2 boot classes. With version 1.4.2 of Java 2, some Toshiba specific code has been moved out of the core classes. Thus, the boot classpath consists of the first six core JAR files listed above plus all files matching the file specification "ibm\*.jar" in the m:/java2/jre/lib directory. Several JAR files in this directory that would not normally be used by a typical POS application have been omitted from the list above, including a debug version of ibmjsseprovider.jar and some ORB-related classes.

The "-FN:" flag is used to override the file name that is used when the file is created on the RAM disk. If this flag is not specified, then the first file name on the list is used to determine the name used when the file is created on the RAM disk. In either case, only the file name portion is used (any path information is ignored). Also, if the name portion of the file name is longer than eight characters or the extension portion is longer than three characters, the file name or extension will be truncated, as appropriate, in an attempt to produce a valid file name. The file name specified by the -FN: flag is only validated for length, so it is possible to specify file names that are invalid or conflict with other files in the RAM disk preload archive.

It would be possible to include all file names longer than 8.3 characters in the list above without specifying the -FN: flag, because the first eight characters of each file name are unique. However, the -FN flag was used to specifically indicate what the file names on the RAM disk would be in order to allow the boot classpath to be set up properly.

## Creating a BAT File

There are several steps involved in executing TOFCreat for multiple applications. It is recommended that a BAT file be used to perform the procedure in order to reduce the chance of omitting a step.

**Note:** The `-classpath` flag should be used to provide a classpath specific to each application in order to prevent class definition conflicts between different archives used by different applications.

The following sample BAT file performs TOFCreat on two applications for Java 2 and then rebuilds the terminal load shrink.

```
REM **** Run TOFCreat on the parent application ****
java2bin:java -cp
m:/app0Files/app0.jar
com.ibm.OS4690.TOF.TOFCreat m:/tofFiles/app0.prp

REM **** Run TOFCreat on the child application ****
java2bin:java -cp
m:/app1Files/app1.jar
com.ibm.OS4690.TOF.TOFCreat m:/tofFiles/app1.prp

REM **** Rebuild the RAM Disk preload image ****
REM
REM **** Note: Prior to executing this BAT file ****
REM **** the preload list file should have been ****
REM **** updated to include the archives that ****
REM **** should be placed on the RAM disk ****
ADXTRMOL

REM **** Rebuild the terminal load shrink ****
ADXRTCCCL
```

---

## JavaTOF Procedural Enhancements

The TOFStartApp class, used to start applications from JavaTOF archives, has been enhanced.

- The JavaTOF properties file, used prior to 4690 OS V3R2 to start up the application, is no longer required to be listed in the classpath or to be contained in an archive. The JavaTOF properties file can be located on the RAM disk or at any location accessible to the terminal.
- If fonts have been preloaded to the RAM disk for Java 2, then TOFCreat prepends the RAM drives to the `java.awt.fonts` property, which the AWT uses to locate fonts. Also, TOFCreat prevents the JVM from looking for fonts in the default font directory (`m:\java2\jre\lib\fonts`). This prevents the AWT from using fonts on the controller, which can cause application problems if the controller is unavailable. (The font files are kept open and frequently referenced during the lifetime of an application).
- JavaTOF does additional checking of user settings when it is run. In particular, JavaTOF checks many of the common JVM properties that contain path information and warns the user if any of the paths refer to remote drives. The list of properties checked include: the application classpath, the boot classpath, and the `java.ext.dirs` property.
- In addition, if the current directory of the JVM, which is set by the `TJAVADEF` or `TJAVA2DEF` logical names, refers to a location on the controller, the Java classpaths are checked for leading, trailing, and double semicolons. This checking is done because of the way JVM works. Path specifications that are "empty" in the Java classpath get resolved to indicate the current directory. If this condition is found, the user is warned because referring to a directory on the controller, when loading classes, can slow down the loading process and put more stress on the controller.

Because of these changes, a user typically receives one or more warnings the first time they attempt to run a JavaTOF application using a setup created prior to 4690 OS V3R2. A brief error message is logged to the `System.out` stream in Java, and a more detailed message is written to the `System.err` stream. In particular, when running Java 2 applications, the default location setting for `java.ext.dirs` is `1:\java2\jre\lib\ext`. Because this refers to a drive on the controller, a JavaTOF warning is generated. If



your application **does not** require any classes or files from this directory, it is recommended that you set this property to an empty string. If your application **does** require classes from this directory (specifically input method classes and data for DBCS languages), the files should be placed in the terminal RAM disk preload file and `java.ext.dirs` should be set to point to the appropriate RAM drive.



---

## Chapter 15. Using the Multiple File Archiver Utility

ADXNSXZL is a multiple file archiver with built-in compression and decompression. It provides a simple and convenient means of copying large files onto a diskette as efficiently as possible. It is not compatible with the 46x0 Compress/Decompress utility and is intended only for command line use.

This utility is built into the installation and migration tools used by the operating system and is also used by the Apply Software Maintenance procedures to minimize the number of diskettes that maintenance requires. Other uses might include archiving a subdirectory before replacing some files for testing purposes. To receive a helpful reminder of the command line options, you can start the application as follows to receive minimal documentation.

<b>Format:</b> ADXNSXZL
----------------------------

---

### Compressing, Combining, and Archiving Files

To create a combine file, you must specify a name for the combine file and which file (or files) you want to be included. Additional files can be added later.

<b>Format:</b> ADXNSXZL C TEST.DAT ADXCSLCF.DAT
--

This example creates a file called TEST.DAT, which contains a compressed version of the dump file ADXCSLCF.DAT.

Efficiency of compression varies greatly depending upon the contents and the size of the file to be compressed. Large dump files compress very well. In contrast, there is negligible compression on small text files. It is possible that the resulting combine file is actually larger than the original uncompressed files. However, in general, ADXNSXZL recognizes if a file is unlikely to benefit from compression, and in this case, simply adds the file to the combine file without attempting to compress it. In particular, ADXNSXZL never tries to compress files that are smaller than 2 KB in size. This provides some optimization of speed versus compression.

Alternatively, if you wanted to compress all the files in ADX\_UPGM: you could use the following command:

<b>Format:</b> ADXNSXZL C MYFILES.DAT ADX_UPGM:**
--

The resulting combine file, MYFILES.DAT, contains all the files in ADX\_UPGM and the original files are left untouched.

When called with the C parameter to create a combine file, ADXNSXZL first checks for the existence of the combine file. If it already exists, an error is reported.

If you want to add files to an already existing combine file, you must use the A parameter.

**Format:**

```
ADXNSXZL A MYFILES.DAT C:\TEST.LOG
```

This example adds a compressed version of the file C:\TEST.LOG to the existing combine file MYFILES.DAT. If the combine file is missing, an error is reported and no action is taken.

Wildcards can also be used with the A parameter to add files to an existing combine file.

**Format:**

```
ADXNSXZL A MYFILES.DAT ADX_UDT1:*.DAT
```

This example adds all files with an extension of DAT in the subdirectory ADX\_UDT1 to the existing combine file MYFILES.DAT without affecting the original files in ADX\_UDT1.

---

## Mapping the Combine File

To determine which files are contained within a given combine file, it is necessary to map the combine file, using the M parameter.

**Format:**

```
ADXNSXZL M TEST.DAT
```

This example lists the contents of the combine file TEST.DAT and reports the original sizes, time stamps, and other information. The following is an example of the output:

**Format:**

```
ADXNSSLG.DAT 468663 1 4-05-1995 15:23 [61%]
```



This example shows that the combine file only contains one compressed file, ADXNSSLG.DAT, that had an original size of 468 663 bytes, a distribution attribute of 1 (local file), and a time stamp of 3:23 PM on the 5th of April 1995. It also lists to what extent the file was compressed as 61% (the compressed version of ADXNSSLG.DAT was 61% of the original size).

The final symbols reflect whether the compressed version of the file is split across several combine files. If the size option for add and compress is not used, this always shows three solid blocks, indicating that the compressed version of the file is contained completely within this combine file. The final symbols are shown below.



Through the use of the size option, at times a compressed file does not fit completely within a single combine file, then other symbols are shown in this space, as follows:



compressed file is contained completely within this combining file



only the beginning of the compressed file is included



only a middle portion of the compressed file is included



only the end of the compressed file is included

If the complete file is not included within the combine file, you need to look in other combine files to find the remaining portions of the compressed file before you are able to split out the original files.

---

## Splitting Files Out of a Combine File

To restore files to their original format from a combine file, you must use the split parameter.

**Format:**

```
ADXNSXZL S TEST.DAT ADX_UPGM:
```

This restores all the files compressed in TEST.DAT to the subdirectory ADX\_UPGM:.

Alternatively, you can request individual files to be split out of the combine file by adding the file name as an optional parameter.

**Format:**

```
ADXNSXZL S TEST.DAT ADX_UPGM: AMCTESTF.DAT
```

This restores just the file AMCTESTF.DAT from the combine file TEST.DAT into the subdirectory ADX\_UPGM:.

The subdirectory is not an optional parameter. If you want to restore files into the current subdirectory, then use the following format:

**Format:**

```
ADXNSXZL S TEST.DAT .\
```

This relies upon the fact that '.' is the current directory. Also, the subdirectory must have a colon (':') or a back slash ('\') at the end of its name. The following is an example of an error:

**Incorrect Format:**

```
ADXNSXZL S TEST.DAT C:\ADX_UPGM (missing trailing back slash)
```

This reports errors for each of the files it attempts to split out of the combine file. Because it tries to create the target file by adding the file name to the directory name as given, it results in an attempt to create files such as C:\ADX\_UGPMAMCTESTF.DAT, which is a file name that is not valid.

To correct the last invocation, a back slash should be appended as follows:

---

**Correct Format:**

ADXNSXZL S TEST.DAT C:\ADX\_UPGM\ (notice trailing back slash)

---

## Splitting a Given List of Files from a Combine File

As well as splitting all files or a specific file out of a combine file, it is possible to split a chosen list of files. This relies upon the L (list) parameter supported by ADXNSXZL:

---

**Format:**

ADXNSXZL L TEST.DAT ADX\_UPGM: FILES.LST

---

This restores all the files listed in FILES.LST from the combine file TEST.DAT into the subdirectory ADX\_UPGM:. FILES.LST should be a list of file names, with one file per line, followed by a carriage return line feed. This can be created with a text editor.

---

## Log Files

If ADXNSXZL is being used as part of another process, perhaps being called from a BATCH file, then it can be useful to have any messages generated logged to a file rather than displayed to the screen. This can be accomplished by using the optional log file parameter on the split and list commands.

---

**Format:**

ADXNSXZL S TEST.DAT .\ (C:\ADXNSXZL.LOG

---

This splits all the files contained in the TEST.DAT combine file into the current directory and reports all messages to ADXNSXZL.LOG in the root directory of the C: drive.

---

## Return Codes

ADXNSXZL displays messages and progress indicators for most of the common functions provided. However, for functions involving split combine files, it often relies upon return codes to report progress and errors.

These return codes are not visible when calling ADXNSXZL directly from the command line. They can be detected through the use of IF ERRORLEVEL when ADXNSXZL is invoked from a BATCH file.

The return codes are:

- 0**      Good
- 1**      Incorrect parameters
- 2**      File is not valid or is corrupted.
- 3**      Combine file is corrupted.
- 4**      File name/directory is not valid.
- 5**      Unable to allocate storage.
- 6**      File already exists.

- 7**      Combine file is not valid.
- 8**      Out of disk space.
- 9**      Temp file is not valid.
- 10**     List file is not valid.
- 11**     Log file is not valid.
- 99**     New combine file is required.





---

## **Part 3. Applications (designing and using)**



---

## Chapter 16. Designing applications with 4680 BASIC

The operating system is a protected-mode operating system. An application can only access code and data areas that are part of that application. Also, because of file security capabilities, the application can only access files that it has created or been designated to access. Memory requirements vary depending on your application's objectives and design. You can make an application a single module or split it into several load modules and chain from one to the other.

See the *4680 BASIC: Language Reference* for the character set, variable types, commands, and syntax rules. You can use a text editor to write your 4680 BASIC applications.

The operating system limits access to files and operating system functions. Each file and function is assigned security attributes that limit its use according to the authorization level of the user. The system maintains a system authorization file, ADXC SouF.DAT, that defines these authorization levels. For more information on the system authorization file and file protection, see "System authorization" on page 282 and "Protecting files" on page 33.

For additional information on how to use the functions of your system, see the *4690 OS: User's Guide*.

---

### Types of applications

The store controller operating system supports two types of applications: interactive and background. An *interactive* application uses the store controller keyboard and display or auxiliary console to directly communicate with the operator. A *background* application communicates directly with the operator through the BACKGROUND APPLICATION CONTROL panel. This panel is a background application control panel used for communication between the operator and background applications.

Using a window, you can run several interactive applications at the same time. Each application runs until it either finishes or operator input is required. When input is required, the application waits for further input before continuing.

### Interactive applications

Interactive applications fall into two categories: primary and secondary. The *primary application* is the main application that runs in your store's normal operating environment. *Secondary applications* are applications that are selected from the SECONDARY APPLICATION menu. This menu is reached by choosing the second option on the SYSTEM MAIN MENU.

Each interactive application is assigned a *window*. Windows enable each application to execute as if it had actual control of the screen and keyboard. The *4690 OS: User's Guide* explains windows.

### Background applications

Background applications are non-interactive store controller applications. Some background applications start or stop running when the system is IPLed or when the acting master store controller or acting file server is changed on a LAN (MCF Network) system. The operator or the host must start other applications that do not start automatically.

There are two kinds of background applications: *permanent* and *temporary*. You define *permanent* background applications when designing your store's configuration. You define the name of the background application, the parameters passed to it, the text, and whether it begins at IPL by using the configuration panels.

*Temporary* background applications are started from the host site or from your application using the ADXSTART interface (see "ADXSTART" on page 280). HCP, if begun from the host site, runs as a

temporary background application. The operator can remove temporary applications if they are not active by pressing **Clear** while viewing the BACKGROUND APPLICATION CONTROL panel.

You can determine the status of your background applications by displaying the BACKGROUND APPLICATION CONTROL panel and specifying the application name. This panel gives you the status of the executing background application, the parameters passed to it, and any message sent by the background application to the operator. See the *4690 OS: User's Guide* for information on BACKGROUND APPLICATION CONTROL.

Your applications update the BACKGROUND APPLICATION CONTROL status panel by writing messages to it periodically. Use a function called Display Background Application Status to write these status messages. This function is described in “Using Application Services” on page 293.

---

## Application size

The operating system has several size restrictions on memory and disk space that your programs must meet. The restrictions are different for the terminal and the store controller.

When planning your applications, you should know how much disk space is available to you. The operating system keeps track of the amount of space already used and the amount available on your disk. You can determine this information by using the CHKDSK or DIR commands.

The CHKDSK command has options through which you can get a report of how much total disk space you have, how much of it contains files, how much space is still available, and how much space, if any, is considered defective.

The DIR command gives you a listing of the files on your disk, how much space they occupy, and how much space remains available. For more information on these commands, see the *4690 OS: User's Guide*.

## Memory models

BASIC supports three different memory models: large (for store controllers) and big and medium (for terminals). With large or big memory models, your application can have more than 64-KB of code and more than 64-KB of *heap* data (see “Data size” on page 279). Big memory models allow up to one 64-KB segment of *static* data, while large memory models allow multiple 64-KB segments of static data. Medium memory models require that all *stack*, static, and heap data reside within one 64-KB segment.

Use the compiler directive %ENVIRON to specify whether your application is to execute in the store controller or in the terminal. For more details on memory models, see the memory allocation chapter in the *4680 BASIC: Language Reference*.

## Code size

Each 4680 BASIC program module that you compile produces an object module (OBJ file). An object module contains a code segment and a data segment.

Each code segment can have a maximum of 64 KB. When you compile a program module, the compiler lists the number of bytes required for the code segment. When you link your program modules together to form a load module, the link editor lists the code size (which is the total number of bytes for all of the code segments of the modules).

The maximum code size supported by the link editor is 2 MB. If your code size exceeds the amount of memory you have available, you can consider chaining from one load module to another to decrease the memory required by each load module. See “Chaining applications” on page 313 for more information on chaining.

Each code segment requires a code segment name. You can use a maximum of 200 code segment names in a load module for the terminal. The 4680 BASIC compiler uses up to five of these code segment names for each load module. Each object module uses one code segment name. Therefore, you can have up to 195 object modules per load module plus five code segment names used by the compiler for a terminal application. There is no limit to the number of object modules in a load module for an application in the store controller.

## Data size

The total data area for a load module is composed of three elements: static data, heap data, and stack data.

The terminal medium memory model default data size is almost 64 KB. If you want to change the value of the terminal data size, use the `DATA[MAX[]]` option on the `LINK86` command. The maximum data area size is almost 64 KB. To define the maximum data area size, use the `DATA[MAX[FFF]]` option.

The terminal big memory model and store controller default data area size is the sum of the initial 8-KB heap plus the load module static data plus the stack. The data area size changes dynamically with the size of the heap.

The *static data* area is used to hold your integer and real variable data for your code modules. It also contains a pointer for each string variable and a pointer for each array definition. In addition, it contains these same types of data for the shared runtime library, common variables, and overlay variables.

When you compile a program module, the compiler lists the number of bytes of static data required. When you link your program modules together to form a load module, the link editor lists the static data size, which is the total number of bytes of static data for all of the program modules. The maximum static data area for a medium model terminal load module is 64 KB minus the stack and minus the heap. The maximum static data area for a big memory model terminal load module is 64 KB. The maximum static data area for a store controller load module is one million bytes.

For medium memory models (terminals only), there is one data area whose size can be no larger than 64 KB minus 16 bytes. (This is the default.) This area contains the stack (2K), the static data, and the heap data. Big memory models allow up to 64 KB of static data. Large memory models allow multiple 64 KB of static data, up to 1 MB. Both large and big memory models allow up to 64 KB of stack space.

The *heap data* area is where variable length items are kept. These items include all string variables, array elements, application file and device buffers, and other runtime subroutine library data. In the terminal, the size of the heap is determined by the number of bytes remaining in the data area after the stack and the static data are allocated. In the store controller, and in the terminal big memory model, the size of the heap is determined dynamically. The program requests heap space as needed.

You can determine the amount of available heap space from the application. The `FRE` function indicates the total number of bytes available everywhere in the heap, and the `MFRE` function indicates the largest number of contiguous bytes available anywhere in the heap.

The *stack data* area is used to pass parameters from one function or subprogram to another. It is also used by the runtime subroutine library procedures to pass internal data. The stack also contains the caller's return address.

The size of the stack is determined when the application load module is loaded. The loader must be able to obtain the minimum stack size request, which is 128 bytes for the terminal medium memory model, 1024 bytes for the terminal big memory model, and 2560 bytes for the store controller, or the application load does not succeed. The loader obtains as much memory as possible for the stack, up to the maximum size requested.

The runtime stack in the BASIC terminal medium memory model occupies the first 2 KB of the data segment. This stack size is fixed and cannot be changed.

The maximum stack size for big and large memory model applications is 64 KB. Check the LINK86 output messages for the default maximum stack size defined. You can change this with the LINK86 STACK[MAX[]] option. In order to leave more space for dynamic data (the heap), the stack should be no larger than is necessary for the application to run correctly. Refer to Chapter 9, “Using the Linker Utility and the POSTLINK Utility,” on page 207 for more information on STACK and other link options.

## File size

Depending on your needs, you can decide to keep the current default size of system files or change the size. Changing file size can help you manage your storage more efficiently. The *4690 OS: User's Guide* tells you how to change file size when performing controller configuration.

You do not have to change system file sizes. When the system is IPLed, the operating system checks the size of your store controller dump file. If the store controller system dump file size is not large enough, file size is automatically increased. The terminal dump file size is not automatically adjusted.

---

## Application priorities

The operating system runs all controller foreground applications and all terminal applications at priority 5. Background controller applications can run at any priority between 1 (high priority) and 9 (low priority). For most systems, background applications should be run at priority 5, which is the same priority as any foreground applications. Applications are scheduled to run as follows:

- If several applications are running at the same priority, then the operating system uses the *time-slice method*. The time-slice method maintains a queue of application requests and enables each application to process for a certain time period and interrupts execution, so the next application can process.
- The operating system allows the application with the highest priority to execute until one of the following conditions is met:
  - The application has completed execution.
  - The application must wait for access to a resource such as a file.
  - Another application with a higher priority is executed.

## File access priorities in terminal applications

You can specify priorities for file access in a terminal application using the reserve word, PRIORITY. PRIORITY is a reserved word that is valid only in terminal applications. You can specify it with the OPEN or CREATE statements.

When you specify PRIORITY, the operations of that file are performed before requests from other terminal files. For example, if you create a file using PRIORITY, reading from and writing to that file takes priority over other pending terminal file requests without PRIORITY.

**Note:** All terminal file requests have a higher priority than store controller file requests.

---

## Starting a background application

This section describes functions that can start background applications. These functions are contained in the runtime library ADXACRCL.L86.

### ADXSTART

This function starts a background application using the operating system.

This function is declared as follows:

```

FUNCTION ADXSTART (NAME$,PARM$,MESS$) EXTERNAL
INTEGER*2 ADXSTART
STRING NAME$,PARM$,MESS$
END FUNCTION

```

where:

**NAME\$ =** Name of background application to be started (without R::) (maximum length = 21 bytes)  
**PARM\$ =** Parameters for the background application (maximum length = 46 bytes). Note the system automatically passes BACKGRND as the first parameter.  
**MESS\$ =** Initial message to be displayed on the background status panel (maximum length = 46 bytes)

The function is invoked as:

```
return.small=ADXSTART (pname$,iparm$,imess$)
```

This function returns a zero if a command was issued to start the application. Table 25 on page 281 shows return codes and their descriptions.

*Table 25. ADXSTART return codes*

Return Code	Description
0	Function completed successfully.
-1000	Function code (for example, first parameter) is not valid.
-1001	Buffer address or length not valid.
-1170	Application name missing or not valid.
-1171	All background application list entries in use.
-1172	Maximum number of background applications already active.
-1175	Invalid parameter.
-1176	Internal error (unable to open driver).

## ADXPSTAR

This function starts a background application at a specified priority. To better understand ADXPSTAR and its priority, you should become acquainted with how the operating system schedules applications.

Every application has a priority ranging from 1 (highest) to 9 (lowest) with 5 being the default. If several applications are running at the same priority, the operating system uses a time-slice method to service them. Each application is allowed to run for a certain period and stop, so that the next application can run. The operating system allows the application with the highest priority to execute until one of the following conditions occur:

- The application is complete.
- An application with a higher priority becomes scheduled to run and is executed.
- The application is forced to wait for an external event to end.

For example, if a system is running three applications, one at priority 2 and two at priority 5, then the application at priority 2 executes until it has completed or is forced to wait. If an application at priority 1 is submitted, then the application at priority 2 is preempted. The priority 5 applications would begin execution on a time-slice basis after the higher priority applications had completed execution or entered wait states.

Use the ADXPSTAR function carefully. As the operating system allows the highest priority application to execute until it ends or is preempted, it is possible that a high-priority application could cause your system to neglect all lower priority applications. This occurs if the high-priority application takes a long time to either complete or experiences few wait states.

The ADXPSTAR function allows you only to set the priority of the background applications. All previous applications and any applications initiated by the host, whether background or previous applications, run at the default priority 5.

The function is declared as follows:

```
FUNCTION ADXPSTAR (NAME$,PARM$,MESS$,PRIORITY) EXTERNAL
INTEGER*2 ADXPSTAR,PRIORITY
STRING NAME$,PARM$,MESS$
END FUNCTION
```

where:

**NAME\$**

= Name of background application to be started (maximum length = 21 bytes)

**PARM\$**

= Parameters for the background application (maximum length = 46 bytes)

**MESS\$**

= Initial message to be displayed on the background status panel (maximum length = 46 bytes)

**PRIORITY**

= Priority of the background application (value from 1 to 9; default=5)

The function is invoked as:

```
return.small=ADXPSTAR (pname$,iparm$,imess$,iprior)
```

This function returns a zero if a command was issued to start the application. Table 26 on page 282 shows return codes and their descriptions.

Table 26. ADXSTAR return codes

Return Code	Description
0	Function completed successfully.
-1001	Buffer address or length not valid.
-1170	Application name missing or not valid.
-1171	All background application list entries in use.
-1172	Maximum number of background applications already active.
-1175	Invalid parameter.
-1176	Internal error (unable to open driver).
-1177	Priority given is out of range.

---

## System authorization

To ensure system and data security, the operating system requires that each operator using the system be authorized. This authorization is granted through a *system authorization file* named ADXCSOUF.DAT. It contains *operator authorization records*. Each operator using the system must have a record in this file. This record contains the operator identifier (ID), password, group ID, and user ID and specifies which system request keys and menu options the operator ID can use.

When an operator signs onto the store controller console, the system verifies the operator ID and password entered with the ID and password in the system authorization file. If both are valid, the operator is allowed to use those system functions specified for the ID by the operator authorization record.

The system authorization file is placed on your system during installation. The ADXCSOUF.DAT file resides in subdirectory ADX\_IDT1 and has the same file protection as your other application files.



The operating system does not provide operator authorization for terminal signon. The 4680 or 4690 applications provide this function. If you write your own application, it must provide this function if it is needed.

## ADXAUTH

For your operators to sign on and use the system, you must create and maintain authorization records for each one. To add, change, or delete an operator authorization record, your application program must use the function ADXAUTH. The application program using this function should be an interactive application. You can also use this function in a background application, but only with functions that do not require screens to be used (functions 3, 8, 9, and 10). This function can be used only in the store controller and is only in ADXACRCL.L86.

The user ID and group ID for each operator can be set with ADXAUTH. Operators needing access to files in command mode should be assigned according to:

Files in subdirectories	User ID	Group ID
ADX_Ixxx	1	2
ADX_Uxxx	1	3

xxx can be any subdirectory name beginning with ADX\_I or ADX\_U.

Software development functions should be performed in the ADX\_Uxxx subdirectories.

The following example shows how to declare the ADXAUTH function:

```
FUNCTION ADXAUTH (FUNC, OPID$,OPPW$,OPID2$) EXTERNAL
STRING          OPID$,OPPW$,OPID2$
INTEGER*2       ADXAUTH,FUNC
END FUNCTION
```

where:

**FUNC** = The action to be taken.  
**OPID\$** = The operator ID on which an action is to be taken.  
**OPPW\$** = The password for the ID.  
**OPID2\$** = The current operator ID or the template ID depending on the requested function. For FUNC parameter 10, OPID2\$ indicates two IDs separated by a colon. The first is the template, and the second is the OPID whose authorization must not be exceeded.

The following example shows how to invoke the ADXAUTH function:

```
AUTH.RET=ADXAUTH (func.code,operator,password,operid2)
```

AUTH.RET is a 2-byte integer variable that receives the return code from the function.

**Note:** The operating system does not restrict access to your primary and secondary applications. If security is required for these applications, you must provide it in the application. The current operator ID is the only parameter passed to these applications and can be used as part of an application authorization function if required.

### FUNC parameter for operator authorization

The FUNC parameter tells you what action can be taken. The functions you can choose are:

- 1** = Change an operator authorization record.
- 2** = Add an operator authorization record.
- 3** = Delete an operator authorization record.
- 8** = Change password only.

- 9 = Make operator ID have the same authorization as the operator ID specified by OPID2\$. If OPID2\$ does not exist, a new ID is created.
- 10 = This function is the same as FUNC 9 except OPID2\$ contains a template ID and a second ID whose authorization must not be exceeded.

If you choose the ADD or CHANGE functions, the system displays several panels from which you select the system functions the operator can use. The current operator (OPID2\$) can select for another ID (OPID\$) only those functions for which the current operator ID is authorized.

**Note:** You must have at least one ID on your system that is authorized for all system functions.

The current operator ID making a change to a record cannot be the same as the operator ID being changed.

If you try to change an authorization record that does not exist, you receive error code -1010, and your request is handled as an add function. The system creates a record having default authorization. The initial default authorization for the add function makes no system functions available. The operator would be allowed only to sign on and select primary and secondary applications.

If you try to add a record for an operator ID that already exists, you receive error code -1010, and your request is handled as a change function.

If you try to delete an authorization record that does not exist, you receive error code -1010.

The make function enables you to use an operator ID as a template for other IDs. You can create or change other IDs without having to select each system function from the panels. The new or changed IDs have the same authorization as the template ID. However, for FUNC 10, the new ID does not have greater authorization than the ID whose authorization must not be exceeded.

If the template ID you specify for the make function does not exist, error code -1011 is returned and default authorization is used as the template. A new ID having default authorization is created.

Table 27 on page 284 shows the FUNC parameter return codes and their descriptions.

*Table 27. FUNC Parameter for Operator Authorization Return Codes*

Return code	Description
0	Function completed successfully.
-1000	Unknown function code.
-1001	File not found.
-1002	Error reading or writing the disk.
-1003	OPID\$ not specified.
-1004	No password specified.
-1005	No OPID2\$ (only when two OPIDs required).
-1010	One of these conditions exists: <ul style="list-style-type: none"> <li>• Add function was handled as change because ID already existed.</li> <li>• Change function was handled as an add because ID was not found.</li> <li>• Delete requested for ID that was not found.</li> </ul>
-1011	Function handled as requested using default authorization because OPID2\$ was not found.
-1020	Memory could not be allocated to process authorization.

### **OPID\$ parameter—Operator ID**

This parameter indicates the operator ID to add, change, or delete. The OPID\$ parameter should be a string containing up to nine ASCII alphanumeric characters. There should be no leading blanks. Leading blanks and zeros are considered part of the operator ID. Trailing blanks are ignored.

### **OPPW\$ parameter—Operator Password**

This parameter is the password for functions 1, 2, 8, 9, and 10. The OPPW\$ parameter should be a string containing up to eight ASCII alphanumeric characters. This parameter should not contain leading blanks. Leading blanks and zeros are considered part of the password. Trailing blanks are ignored.

### **OPID2\$ parameter**

For FUNC parameters 1 and 2, the OPID2\$ parameter should be a previously defined operator ID. For FUNC parameter 9, this parameter indicates the operator ID to be used to set the authorization for the ID specified by OPID\$. The OPID2\$ parameter should be a string containing up to nine ASCII alphanumeric characters without any leading blanks. Leading zeros are considered part of the operator ID. Trailing blanks are ignored. For FUNC parameter 10, OPID2\$ contains two IDs separated by a colon. The first is the template ID, and the second ID is the ID whose authorization must not be exceeded (this can be the currently signed on ID). Both IDs should be strings containing up to nine alphanumeric characters.

For example:

```
"1234" + ":" + "4567"
```

For other FUNC parameters, this parameter is ignored.

## **ADXCRIPT**

The operating system uses one-way encrypted passwords. The system encrypts passwords before storing them in the system authorization file. When an operator signs on, the password entered is encrypted and the encryption code is compared to the encrypted password code in the operator's authorization record. If the two codes match, the operator is allowed to sign on. If your application files contain passwords, your applications should encrypt the passwords. You can use the following subprogram to encrypt an eight-character ASCII password:

```
SUB ADXCRIPT (RETCODE, PWIN$, PWOUT$) EXTERNAL
STRING      PWIN$,PWOUT$
INTEGER*2   RETCODE
END SUB
```

where:

#### **RETCODE**

= Contains the return code from the call.

#### **PWIN\$**

= Contains the password to be encrypted.

#### **PWOUT\$**

= Receives the encrypted value.

The PWIN\$ parameter should be a string containing up to eight ASCII alphanumeric characters with no leading blanks. Leading zeros are considered part of the password. If this parameter is missing, error code -1100 is returned.

The PWOUT\$ parameter returns an eight-character string containing ASCII characters that represent decimal digits.

You can use this subprogram in both the store controller and the terminal. It is in ADXACRCL.L86, ADXUCLTL.L86, and ADXUCRTL.L86.

---

## ADXDATE (retrieving a 4-digit year system date)

You can call the following subprogram to retrieve a 4-digit year system date:

```
SUB ADXDATE(RC, BUFFER) EXTERNAL)
    INTEGER*4 RC
    STRING BUFFER
END SUB
```

This routine writes the system date into the caller's string variable. The date is in the form YYYYMMDD.

where:

**RC** = Contains the return code from the call:

- 0** = Retrieving the system date was successful
- 1** = Either the string variable cannot be expanded to hold the string, or an operating system error has occurred

Use this subprogram in both the store controller and the terminal. It is located in these executables:

- ADXACRCL.L86
- ADXUCLTL.L86
- ADXUCRTL.L86

---

## Communicating between applications

Your applications can communicate with each other in three ways. Any application can write information to or read information from a file. Any application can also use *pipes*. Pipes are file-like structures in memory used for communicating between applications. Exchanging data using pipes is much faster than with disk files. A third method is for communicating with Java applications. This method uses the 4690 Java-BASIC API.

### Using pipes

Use pipes to communicate data that can be recreated by the application writing the data; if there is a power line disturbance the store controller is IPLed, and the contents of the pipe are lost. Use disk files for data that cannot be recreated.

### An overview of pipes

Pipes are used in the following types of application communication:

- Between applications at the store controller
- Between applications from one terminal to another terminal or the store controller
- Between applications from one node to another node on a multiple store controller system

A pipe is an area of memory that is like a sequential file. Your application creates a pipe by using the CREATE command and specifying a pipe name. A pipe name has the same characteristics as a file name. The identifier pi: must precede the pipe name and no extension (xxx) is allowed.

More than one program can write to a pipe, but only one reads data from it. Because pipes are used this way, two pipes must be used to communicate both ways between applications.

Pipes have security restrictions similar to those used for files. The application creating the pipe owns it, and the system saves the user and group ID of the creator. The system creates and opens the pipe in read-write mode unless your application specifies otherwise.

A pipe is temporary. When the last application that has access to a pipe closes that pipe, the pipe is deleted. If a program using a pipe is terminated by the operating system (not a normal application termination), the operating system automatically closes the pipe. See the *4680 BASIC: Language Reference* for more detailed information on pipes.

A pipe can be created with a zero-length buffer size for use as a simple semaphore. A READ operation of a semaphore pipe obtains the semaphore and a WRITE releases it. If another process has obtained the pipe previously, the READing process waits until a WRITE to that pipe has been performed. Write operations, on the other hand, never wait; even if the pipe was released previously, the call returns without an error. To create a semaphore pipe in a BASIC application, omit the BUFFSIZE parameter and specify BUFF as 0 in the CREATE statement.

## Pipe Routing Services

Pipe Routing Services is a facility of the operating system that enables applications to exchange data with applications in other store controllers or terminals by using pipes.

To exchange data with other store controllers or terminals, you must use the pipes created through Pipe Routing Services.

Identify these pipes using pipe IDs. A pipe ID is a letter between A and Z. Make each ID in a store controller or terminal unique. Each store controller or terminal can have up to 26 IDs (A to Z). If you have several applications running in a store controller at once, each must use a different pipe ID.

One of three runtime libraries (one for the controller and two for the terminal) contain these functions, depending on whether you are requesting services for the store controller or the terminal.

The large memory model controller library is ADXACRCL.L86.

The medium memory model terminal library is ADXUCRTL.L86.

The big memory model terminal library is ADXUCLTL.L86.

The system has separate functions for the terminals and the store controllers. The function names are as follows:

PRStyyy

where:

**x** = T for functions used in the terminal  
**x** = C for functions used in the store controller  
**yyy** = CRT for the create pipe function  
= WRT for the write function  
= INT for the initialization function

To prepare for receiving data through a Controller Pipe Routing Services pipe, call this function:

```
FUNCTION PRStCRT (IONUM,SIZE%,PIPEID) EXTERNAL
INTEGER*2 PRStCRT
INTEGER*2 IONUM,SIZE%
STRING PIPEID
END FUNCTION
```

where:

<b>IONUM</b>	An I/O session number used for normal opens and creates. Use the same number for waits and reads that follow.
<b>SIZE</b>	The number of bytes to allocate for the pipe size. Maximum pipe size for pipes created by terminal applications is 240 bytes. If you inadvertently define a size larger than the maximum allowed, the pipe size is limited but no error message appears. The maximum pipe size for pipes created by a controller application is 65,536 bytes.
<b>PIPEID</b>	A character between A and Z. (Lowercase is forced to uppercase before it is used by this function.)

Table 28 on page 288 shows the two-byte integers the PRStCRT function returns.

Table 28. Function PRSxCRT Two-Byte Integers

Integer	Description
0	Good completion
-1000	Invalid pipe ID
-1001	Pipe already exists

**Note:** Your calling program should have an ON ERROR routine to handle normal pipe creation runtime errors.

Before reading data from a Pipe Routing Services pipe, your application should use the WAIT statement to obtain notification that there is data in the pipe.

**Note:** If a terminal application issues a WAIT for a Pipe Routing Services pipe, you should be aware that the terminal sends a message to the controller by way of the TCC Network. This message tells the controller that the terminal is waiting for data to become available in the pipe. If the WAIT ends because the timeout interval specified on the WAIT is exhausted, the terminal sends another message to the controller by way of the TCC Network. This message tells the controller that the terminal is no longer waiting for data in that pipe. Repeatedly issuing WAITs for Pipe Routing Services pipes with short timeout intervals in the terminal results in a large number of TCC Network messages being sent to the controller. This additional volume of TCC Network messages can have an adverse impact on the controller's response time to terminal requests. Therefore, short timeout intervals on the WAIT should be used only if absolutely necessary. In no case should the timeout interval be less than 100 milliseconds, as this much time is required to send the WAIT message to the controller and to receive a response back from the controller.

When the WAIT statement finishes for the pipe, the application should use the READ FORM# statement to read the data from the pipe. The READ FORM# statement should specify the number of bytes to read according to the type of message written to the Pipe Routing Services pipe. You can use fixed-length or variable-length message types.

For fixed-length message types, the number of bytes read from the pipe must be the same as the number of bytes written to the pipe. Fixed-length messages require that the application writing to the pipe and the application reading from the pipe always use the same number of bytes in their messages.

For variable-length message types, the number of bytes read from the pipe must be less than or equal to the number of bytes written to the pipe.

A variable-length message consists of two parts. The first part is fixed-length and indicates whether there is a second part and the size of the second part. The application writing the variable-length message writes both parts of the message together as one message. The application reading the variable-length message reads the first part of the message by reading a fixed-length message. From the first part of the message the application determines the number of bytes to read for the second part.

The following conditions can result in errors being returned to a terminal application for a READ FORM# statement. See the *4690 OS: Messages Guide* for error information.

- If a READ FORM# specifies more bytes for a message than are actually in the Pipe Routing Services pipe, the system purges all of the data in the Pipe Routing Services pipe.
- If the program issues a WAIT and is notified that data is in the Pipe Routing Services pipe, it is possible for the data to become unavailable before the program can issue the READ FORM#. This can occur if the program that wrote the data into the Pipe Routing Services pipe is ended before the READ FORM# is issued.
- If a program attempts to read a message of more than 120 bytes, an error is returned. Pipe Routing Services pipe messages can contain a maximum of 120 bytes.

Sending data to other terminals or store controllers is a two-part process. First, your application must initialize the writing service. Use the function shown here to perform this initialization.

```
FUNCTION PRSXINT EXTERNAL  
  INTEGER*4 PRSXINT  
END FUNCTION
```

```
  INTEGER*4 PRSNUM
```

```
  PRSNUM=PRSXINT
```

This function returns a four-byte integer. Your application must save this integer for use when writing. You should call the initialization function only one time for each load of the application.

If your controller application calls the PRSCINT function, and if your controller application uses the CHAIN statement to transfer control to another controller application, your application should call the PRSCCLS function before chaining:

```
FUNCTION PRSCCLS EXTERNAL  
  INTEGER*1 PRSCCLS  
END FUNCTION
```

```
  CALL PRSCCLS
```

PRSCCLS releases the resource obtained by the PRSCINT function and makes it available to other programs. If you omit this call to PRSCCLS, eventually this resource could be depleted. PRSCCLS does not return a return code.

After initialization, you can write to another store controller or terminal. Do this using the following function:

```
FUNCTION PRSXWRT (PRSNUM,DEST,BUFFER) EXTERNAL  
  INTEGER*4 PRSXWRT  
  INTEGER*4 PRSNUM  
  STRING  DEST, BUFFER  
END FUNCTION
```

where:

**PRSNUM** = Contains the value returned by the initialization function.

**BUFFER** = Contains the data to be sent. The maximum length of data that controller applications can write to controller pipes or terminal pipes is 120 bytes. The maximum length of data that a terminal application can write to a terminal pipe is also 120 bytes. The maximum length of data that a terminal application can write to a controller pipe is 512 bytes.

**DEST** = Contains the destination address. This is four characters of the form *aaaw*.

The *aaa* indicates the terminal or store controller to which to send. For terminals, *aaa* is the terminal number in ASCII. For store controllers, it consists of 0xy where x and y are characters between C and Z. Thus, for the store controller, *aaa* can range from 0CC to 0ZZ. There are also two special values of *aaa* for store controllers: 0AA and 0BB. Use 0AA when the destination is the master store controller. Use 0BB when the destination is the controller to which the terminal is attached. You can think of these destinations as logical destinations. The destination is associated with the machine performing the function rather than the physical machine that is assigned the address. Where necessary, the operating system switches a destination to another machine when a configuration change occurs that affects the destinations. For example, the system switches destinations when one store controller takes over another store controller's TCC Network. Pipe Routing Services perform the translations of 0AA and 0BB so that your application does not have to actually know the real store controller addresses.



The *w* part of the destination address indicates which pipe to write to in the specified store controller or terminal. It is a pipe ID. It should be one of the pipe IDs used by an application in the destination terminal or controller to create a Pipe Routing Services pipe.

**Note:** If a terminal application is writing to a pipe created by another terminal, both terminals must be located on the same controller. A terminal application cannot write a message across the LAN to a terminal located on a different controller.

Table 29 on page 290 shows the 4-byte integers that are returned by the PRSxWRT function.

Table 29. Function PRSxWRT Four-Byte Integers

Integer	Description
0	Good completion.
-1	Destination pipe is full or does not have enough space available to hold the data being written.
-1010	Invalid destination specified.
-1011	Destination not found.
-1012	Error on write to destination.
-1013	Data length greater than the maximum allowed length.
-1014	Buffer is not valid.

Additionally, you can choose to use the conditional pipe write. This write is identical in every way to the normal PRSxWRT pipe write with one additional function. If the destination pipe is full or does not have enough room left to contain the entire message being written, the write does not wait for room to become available. Instead, the application is given control immediately with a -1 return code. At this point, the application can make a decision to either discard the data being written or to retry the write at a later time. The intended use for the conditional pipe write is for situations where it is undesirable for an application to wait for extended periods of time. To use the conditional pipe write, use the following function:

```
FUNCTION PRSxWRC (PRSNUM,DEST,BUFFER) EXTERNAL
INTEGER*4 PRSxWRC
INTEGER*4 PRSNUM
STRING    DEST, BUFFER
END FUNCTION
```

where:

**PRSNUM** = Contains the value returned by the initialization function.

**BUFFER** = Contains the data to be sent. The maximum length of data that controller applications can write to controller pipes or terminal pipes is 120 bytes. The maximum length of data that a terminal application can write to a terminal pipe is also 120 bytes. The maximum length of data that a terminal application can write to a controller pipe is 512 bytes.

**DEST** = Contains the destination address. This is four characters of the form *aaaw*.

The *aaa* indicates the terminal or store controller to which to send. For terminals, *aaa* is the terminal number in ASCII. For store controllers, it consists of 0xy where *x* and *y* are characters between C and Z. Thus, for the store controller, *aaa* can range from 0CC to 0ZZ. There are also two special values of *aaa* for store controllers: 0AA and 0BB. Use 0AA when the destination is the master store controller. Use 0BB when the destination is the controller to which the terminal is attached. You can think of these destinations as logical destinations. The destination is associated with the machine performing the function rather than the physical machine that is assigned the address. Where necessary, the operating system switches a destination to another machine when a configuration change occurs that affects the destinations. For example, the system switches destinations when



one store controller takes over another store controller's TCC Network. Pipe Routing Services perform the translations of 0AA and 0BB so that your application does not have to actually know the real store controller IDs.

The *w* part of the destination address indicates which pipe to write to in the specified store controller or terminal. It is a pipe ID. It should be one of the pipe IDs used by an application in the destination terminal or controller to create a Pipe Routing Services pipe.

**Note:** If a terminal application is writing to a pipe created by another terminal, both terminals must be located on the same controller. A terminal application cannot write a message across the LAN to a terminal located on a different controller.

Table 30. PRSCWRC function four-byte integers

Integer	Description
0	Good completion.
-1	The destination pipe is full or does not have enough space available to hold the data being written.
-1010	Invalid destination specified.
-1011	Destination not found.
-1012	Error on write to destination.
-1013	Data length greater than the maximum allowed length.
-1014	Buffer is not valid.

## Using the 4690 Java-BASIC API

The 4690 Java-BASIC API is a group of subprograms that enable BASIC applications to exchange data with Java applications. The subprograms construct a request string containing the Java class, method, and arguments, and then passes that request string to Java. The communication mechanism between Java and BASIC is the Java DeviceManager class, which must be instantiated by the Java application. For additional information about the Java DeviceManager, see "Class DeviceManager" on page 540

After Java has processed the request string, the subprograms in the API move any data returned from the Java application into the BASIC application's variables.

**Note:** Java methods called from this environment must be static methods, and all parameters in the method must be Java objects and cannot be a primitive type.

### Other Java-related information in this guide

For more information about:

- Starting a Java application, see "Starting a Java application" on page 311
- Stopping a Java application, see "Stopping a Java application" on page 312
- Designing applications with Java, see Chapter 20, "Designing applications with Java," on page 455
- I/O redirection to Java, see "Using I/O Redirection to Java" on page 539
- The Java IO processor, see "Java I/O Processor Functions" on page 586

There are three runtime libraries that contain the 4690 Java-BASIC API subprograms:

- ADXACRCL.L86, the large memory model controller library
- ADXUCRTL.L86, the medium memory model terminal library
- ADXUCLTL.L86, the big memory model terminal library

The BASIC subprogram shown below opens communications with Java and constructs a request string that contains the name of the Java class and method to be invoked:

```
SUB JavaCall.Initialize.Request (ClassName,MethodName, \
                                TheRequest) EXTERNAL
    STRING    ClassName
    STRING    MethodName
    STRING    TheRequest
END SUB
```

Parameter meanings are:

**ClassName** The Java class name

**MethodName** The Java method name

**TheRequest** TheRequest is the request string that BASIC passes to Java

Call any of the following subprograms to add a parameter to the request string created by **JavaCall.Initialize.Request**:

```
SUB JavaCall.AddParameter.Integer1 (TheRequest,TheParameter) EXTERNAL
    STRING    TheRequest
    INTEGER*1 TheParameter
END SUB
```

```
SUB JavaCall.AddParameter.Integer2 (TheRequest,TheParameter) EXTERNAL
    STRING    TheRequest
    INTEGER*2 TheParameter
END SUB
```

```
SUB JavaCall.AddParameter.Integer4 (TheRequest,TheParameter) EXTERNAL
    STRING    TheRequest
    INTEGER*4 TheParameter
END SUB
```

```
SUB JavaCall.AddParameter.String (TheRequest,TheParameter) EXTERNAL
    STRING    TheRequest
    STRING    TheParameter
END SUB
```

**Note:** You can call multiple subprograms to add multiple parameters to the request string.

Parameter meanings are:

**TheRequest** The string variable that JavaCall.Initialize.Request initializes to specify the Java class and method to be invoked

**TheParameter** A variable or literal of the appropriate data type

Call one of the following subprograms to pass the completed request string to Java. Choose the subprogram according to the data type of the value to be returned by the method you name in TheRequest.

**Note:** If the BASIC application calls one of the following subprograms before the DeviceManager class has been instantiated, the subprogram does not return control to the BASIC application. In other words, the BASIC application is blocked until the Java application has instantiated the DeviceManager.

```
SUB JavaCall.InvokeMethod.ReturnVoid (TheRequest,Exception ) EXTERNAL
    STRING    TheRequest
    STRING    Exception
END SUB
```

```
SUB JavaCall.InvokeMethod.ReturnI1 (TheRequest, ReturnValue, \
                                    Exception) EXTERNAL
    STRING    TheRequest
```

```

        INTEGER*1 ReturnValue
        STRING    Exception
    END SUB

    SUB JavaCall.InvokeMethod.ReturnI2 (TheRequest, ReturnValue, \
                                        Exception) EXTERNAL
        STRING    TheRequest
        INTEGER*2 ReturnValue
        STRING    Exception
    END SUB

    SUB JavaCall.InvokeMethod.ReturnI4 (TheRequest, ReturnValue, \
                                        Exception) EXTERNAL
        STRING    TheRequest
        INTEGER*4 ReturnValue
        STRING    Exception
    END SUB

    SUB JavaCall.InvokeMethod.ReturnString(TheRequest, ReturnValue, \
                                           Exception) EXTERNAL
        STRING    TheRequest
        STRING    ReturnValue
        STRING    Exception
    END SUB

```

Parameter meanings are:

**TheRequest** The request string created by JavaCall.Initialize.Request with optional parameters added by one or more of the JavaCall.AddParameter subprograms. You can reuse a request string indefinitely.

**ReturnValue** The data returned by the Java application to the BASIC application. ***ReturnValue is a different data type in each routine shown above.*** Call **JavaCall.InvokeMethod.ReturnVoid** if there is no return value expected from Java for a specific request string.

**Exception** An empty string after return to the BASIC application if the operation completed without error.

If Java detects an error, Exception contains the exception string returned from Java, which might be more than one message. The message delimiter is a semicolon (;).

If the Java-BASIC API detects an error, it returns an exception string of the format:

```
JAVAAPI.ERR.nnnnnnnn
```

where *nnnnnnnn* might be an OS error number but can also be other information that describes the error.

---

## Using Application Services

Application Services is a group of operating system services that are available to store controller and terminal applications for performing various tasks. Some of these services can be used by both store controller and terminal applications.

### Store controller and terminal application services

This section lists routines that are available to applications at the store controller and the terminal. The function code and parameters for each function are listed, along with an explanation of the function.

#### ADXSERVE (requesting an application service)

Both store controller and terminal applications use a routine called ADXSERVE to request Application Services.

ADXSERVE is located in the following libraries:

- ADXACRCL.L86 for store controller applications
- ADXUCLTL.L86 for big memory model terminal applications
- ADXUCRTL.L86 for medium memory model terminal applications

The routine must be declared as follows:

```
SUB ADXSERVE (RET,FUNC,PARM1,PARM2) EXTERNAL
INTEGER*4 RET
INTEGER*2 FUNC,PARM1
STRING    PARM2
END SUB
```

Invoke the routine using this request:

```
CALL ADXSERVE(RET,FUNC,PARM1,PARM2)
```

where:

**RET** = Is the return code. It is zero if the operation was successful, or negative if the operation failed. Return code values are listed below.

**FUNC** = Specifies which ADXSERVE service is to be called.

**PARM1** = Varies according to the function. See the following function descriptions.

**PARM2** = Varies according to the function that is called.

**Note:** If your controller application calls ADXSERVE, it should also call ADXSERCL (see “ADXSERCL (closing an application service)” on page 321) prior to chaining to another application in order to free system resources.

Table 31. ADXSERVE return codes

Return code	Description
-1000	Invalid function code.
-1001	Buffer or length invalid. (Returned when FUNC=25, 26, 27, or 44)
-1002	Invalid terminal number supplied.
-1003	System unable to obtain a flag.
-1015	Cannot send power-OFF message to remote controller because of non-LAN system. (Returned when FUNC=5)
-1016	Machine to be powered-OFF is not 4690 family or controller/terminal environment. (Returned when FUNC=5)
-1017	Invalid date/time specified in ADXSERVE call. (Returned when FUNC=5 or 44)
-1018	Invalid controller ID specified in ADXSERVE call or not active controller. (Returned when FUNC=5)
-1019	Request issued on LAN system controller that is not the master store controller.
-1020	ABIOS driver open failed. (Returned when FUNC=5)
-1021	ABIOS driver call failed because of a system problem, of invalid time, or Programmable Power has been disabled for this terminal. (Returned when FUNC=5) When a Programmable Power command is issued to a terminal that does not have the proper BIOS or jumper setting (e.g. S1 in Enhanced terminals) to execute this command, this error message (-1021) is returned to the application.
-1022	User logical name not defined/unknown request.
-1023	Programmable Power request is pending. Either Programmable Power has been disabled or an application has set the 'No-IPL' flag in a MOD1 or MOD2. (Returned when FUNC=5)*
-1080	Command blocked by system control function from the screen and keyboard. (Returned when FUNC=2 or 3)
-1081	Terminal failed to respond.

Table 31. ADXSERVE return codes (continued)

Return code	Description
-1100	Requestor not an interactive application. (Returned when FUNC=25 or 27)
-1101	Requestor not a background application.
-1138	Background application already running.
-1139	Background application already stopped.
-1170	Specified program name missing or invalid.
-1173	Memory range error on parameter list.
-1175	Invalid parameter.
-1212	Programmable power command issued for non-469x/SurePOS/TCxWave family or controller/terminal.
-1214	Invalid parameter provided for switching the display screen.
-1301	Keyboard and mouse wait already in progress. (Returned when FUNC=20 or 21.)
-1302	Amount of seconds to wait for inactivity (PARM1) is out of the allowed range. (Returned when FUNC=20.)
-1303	Request only valid when the system is in controller or terminal Java mode. (Returned when FUNC=22 or 23.)
-1304	Request not valid when there is no system console active user. (Returned when FUNC=22 or 23.)

**Note:**

You receive an incorrect function code if a store controller-only function is called in the terminal application.

\* When programmable power is subsequently enabled, the pending power-OFF command is issued if the power-ON time has not passed, or if the power-ON time is 9999.

**Dump system storage:** The Dump System Storage function causes all of the storage in the machine at which the request is made to be dumped to a disk file. Both the application and operating system storage are dumped.

The Dump System Storage function uses the following parameters:

**FUNC** = 1  
**PARM1** = Unused; the value is ignored.  
**PARM2** = Unused; the value is ignored.

**Modify incomplete dump sequence:** This function will boot to the supplementals if there are two dumps within a configured time. To modify an incomplete dump causing this behavior you can add a property to the user properties file c:\adx\_idt1\vx\_ctlpr.dat. The property is dump.loop.detect.seconds=*number*, where *number* is either 0 or the number of seconds to use as an interval.

If the property does not exist, or if you set it to 0, then the function is completely disabled.

**Enable terminal storage retention:** This function enables storage retention on terminals. If this function is called in a non multiple-controller LAN (MCF Network), all the terminals on that TCC Network are enabled. If this function is called in a terminal, only that terminal's storage retention is enabled. If this function is called only from the acting master store controller in a multiple-controller LAN (MCF Network), all terminals on that network are enabled.

If the function is called in the terminal, the effect is temporary. The condition established in the controller overrides the temporary terminal condition. Whenever the terminals receive updates from the controller,

terminal online updates occur when terminal-to-controller communications are interrupted or when system functions are used that require sending new status to a terminal. For example, the daily 1 a.m. status update sent from the controller to all terminals.

**Note:** With programmable power, you can enable storage retention from the Operator Console Facility panels at any controller and set the terminal activity timeout value. After the specified number of inactive minutes, the terminal automatically enters Standby/Suspend mode. This behavior does not apply to the terminal side of a controller/terminal.

The Enable Terminal Storage Retention function uses the following parameters:

**FUNC** = 2  
**PARM1** = Unused; the value is ignored.  
**PARM2** = Unused; the value is ignored.

**Disable terminal storage retention:** This function disables storage retention on terminals. If this function is called in a non multiple-controller LAN (MCF Network), all the terminals on that TCC Network are disabled. If this function is called in a terminal, only that terminal's storage retention is disabled. If this function is called from only the acting master store controller in a multiple-controller LAN (MCF Network), all terminals on that network are disabled.

If the function is called in the terminal, the effect is temporary. See "Enable terminal storage retention" on page 295.

The Disable Terminal Storage Retention function uses the following parameters:

**FUNC** = 3  
**PARM1** = Unused; the value is ignored.  
**PARM2** = Unused; the value is ignored.

**Get Application Status Information:** This function gathers status information and places it in the string variable you specify with PARM2. The information returns in ASCII data format. Use the MID\$ statement to extract selected fields from PARM2. MID\$ references the offset in the PARM2 string from the left, which ensures that your program works properly if PARM2 size is extended.

The Get Application Status Information function uses the following parameters:

**FUNC** = 4  
**PARM1** = Unused; the value is ignored.  
**PARM2** = Specify a string variable.

Table 32 on page 296 shows the data for the store controller application status.

*Table 32. Store controller application status*

Offset	Length	Description
0	4	Store number
4	1	Date format: 1 = m/d/y 2 = d/m/y 3 = m.d.y 4 = d.m.y
5	1	Time format: 1 = h:m 2 = h.m
6	1	Monetary format: 1 = period convention (1,234,970.06) 2 = comma convention (1.234.970,06)

Table 32. Store controller application status (continued)

Offset	Length	Description
7	1	Display type 0 = unknown display (for example CMOS information is invalid) 1 = monochrome display 2 = color display
8	3	Number of terminals configured (001 through 999)
11	4	Store controller ID 00CC through 00ZZ for this store controller
15	2	Reserved
17	2	Controller ID for the master is 00 if not found; otherwise CC through ZZ.
19	3	Printer lines per page 001 through 999
22	1	Number of digits after decimal point 0 or 2
23	1	This controller on an active LAN System = 1; else = 0
24	2	Acting ID is a 2-byte ASCII field: • Controller is on LAN = 1 • Master = 16 • Alternate master = 8 • File server = 4 • Alternate file server = 2 • Or a combination of these (add the values) (for example, alternate master and alternate file server = 11; • Master and file server = 21 • Controller is not on LAN = 00
26	1	Printer associated with console (1 through 8)
27	1	Assigned console (0 through 8)
28	1	LAN Type: X'00' = LAN not installed X'01' = Token Ring X'02' = Ethernet
29	51	Reserved

Table 33 on page 297 shows the data for the terminal application status.

Table 33. Terminal application status

Offset	Length	Description
0	4	Store number
4	1	Date format: 1 = m/d/y 2 = d/m/y 3 = m.d.y 4 = d.m.y
5	1	Time format: 1 = h:m 2 = h:m
6	1	Monetary format: 1 = period convention (1,234,970.06) 2 = comma convention (1.234.970,06)
7	3	Terminal number (001 through 999)

Table 33. Terminal application status (continued)

Offset	Length	Description
10	1	Terminal online/offline 0 = offline 1 = online
11	1	Storage Retention 0 = disabled 1 = enabled
12	24	Default Application Name <ul style="list-style-type: none"> <li>• device name — 8 character maximum (:)</li> <li>• file name — 8 character maximum (.)</li> <li>• file extension — 3 character maximum</li> </ul>
36	1	Number of digits after decimal point 0 or 2
37	1	Terminal powered ON/OFF 0 = on 1 = off — Always on for Mod1; can be on or off for Mod2
38	1	Not applicable
39	1	Backup (loop) 1 = Loop in backup 0 = Not in backup
40	1	System Display 1 = Display named ANDISPLAY is the system display 2 = Display named ANDISPLAY2 is the system display 3 = Display named VDISPLAY is the system display 4 = Display named VDISPLAY2 is the system display 5 = Display named ANDISPLAY3 is the system display
41	1	0 = Mod1 1 = Mod2
42	3	Partner terminal address
45	2	Current store controller ID
47	1	Video Display Adapter 0 = 4683 Video display feature A 1 = VGA
48	1	Application Environment 0 = Running in a terminal 1 = Running in a controller/terminal
49	2	Hard Totals NVRAM Size 01 = 1 KB Hard Totals (4683) 16 = 16 KB Hard Totals (4693/4694) 99 = >99 KB Hard Totals (NVRAM is shown as 99 rather than the actual when the size of NVRAM is greater than 99 KB due to the string field size restriction).
51	1	Terminal Type 1 (4693) 2 (4694) 3 (4683) 4 (4684) 5 (SurePOS 300/700 or TCxWave 6140 Series)
52	1	Supporting Operating System 1 (4690 OS V2 or higher) 2 (4690 Terminal Services for DOS)



Table 33. Terminal application status (continued)

Offset	Length	Description
53	1	Printer Device Type X'00' — Pre-4610 printer attached X'30' — 4610 printer attached
<b>Note:</b> If Device type is X'00', offset 54 through offset 57 are also X'00'.		
54	1	Printer Device ID X'00' — Model TI1 or TI2 (Impact DI/Thermal CR)
55	1	Printer Features Bit 0 (X'01') set to 0 — No MICR is present Bit 0 (X'01') set to 1 — MICR is present Bit 1 (X'02') set to 0 — No check flipper is present Bit 1 (X'02') set to 1 — Check flipper is present Bit 2 (X'04') set to 0 — SBCS (single-byte character set) printer Bit 2 (X'04') set to 1 — DBCS (double-byte character set) printer
56	1	Printer Command Set in use X'00' — Command Set version 1
57	1	EC level of code loaded in the printer
59	1	Java redirection information Bit 7 (0x80) set to 1 — I/O Processor redirected Bit 6 (0x40) set to 1 — ANDISPLAY1 handler installed Bit 5 (0x20) set to 1 — ANDISPLAY2 handler installed Bit 4 (0x10) set to 1 — Cash Receipt Station monitor installed Bit 3 (0x08) set to 1 — Cash Receipt Station handler installed Bit 2 (0x04) set to 1 — Document Insert Station handler installed Bit 1 (0x02) set to 1 — Summary Journal Station handler installed Bit 0 (0x01) set to 1 — Magnetic Stripe Reader handler installed
62	1	Terminal keyboard type ' ' — (blank or 0x20) Uninitialized, the keyboard is not yet determined '1' — 50 key keyboard '2' — 133 key keyboard '3' — ANPOS keyboard '4' — 4820 keypad '5' — Keyboard V (DBCS) '6' — Keyboard VI (DBCS) '7' — Modular 67 key keyboard '8' — Modular ANPOS keyboard 'A' Modular 67 key keyboard with display '?' - Undetermined, a keyboard not in this list is attached

**Programmable power:** Programmable power enables terminal or controller applications to issue program calls to enable or disable storage retention in terminals. It also provides the ability for controller applications to issue program calls to power off a controller, to power off a terminal or to place a terminal into a low power state (sleep mode).

**Note:** Programmable power does not apply to the terminal side of a controller/terminal.

When the application calls the power-off function, it specifies the day and time that the power is to be turned back on. The time must be specified in the international format. A 0000 or 2400 is accepted as midnight. The day of month is also specified and must meet either of the following criteria:

- If the day and time are prior to the current day and time, the day must be valid for the next calendar month.

- If the day and time are after the current day and time, the day must fall within the current month.

**Note:** Using 9999 for the time indicates that a power down is to be done without enabling a power-on time.

The maximum time that a controller or terminal can be programmed to wait before powering on is one month plus or minus a few minutes due to clock variances.

*Power off a remote controller:* This function is used to power off a remote LAN (MCF Network) controller. This function can be invoked on a controller from any supported family (personal computer or 469x or SurePOS 700 series). The controller on which this function is invoked must be the master controller in a LAN system. The remote controller being powered off must be in the 469x or SurePOS 700 family to have the programmable power capability. The day, hours, and minutes in the PARM2 string are the power-on time.

**Note:** Some machines do not support power-on time, so they are only able to power off. To specify that the controller power off remote machines, enter the time in the PARM2 parameter. Controllers cannot be put into a standby/suspend mode, only powered off. Controllers that are part of a controller/terminal accept this command regardless of the activity on the terminal side.

This function uses the following parameters:

**FUNC** = 5  
**PARM1** = 1  
**PARM2** = DDHHMMCC  
**DD** = is the day of month (01–31)  
**HH** = The hours (00–24, 99)  
**MM** = The minutes (00–59, 99)  
**CC** = The controller ID (CC through ZZ)

*Power off a remote terminal:* This function is used to power off a remote terminal. This function can be invoked on a controller from any supported family (personal computer or 469x or SurePOS 700 series). The controller that invokes this function must be the master controller if on a LAN (MCF Network) system, or from the stand-alone controller running TCC for terminals. The remote terminal being powered off must be in the 469x or SurePOS 300/700 family to have the programmable power capability. The day, hours, and minutes in the PARM2 string are the power-on time. The remote terminal being powered off must be in the 469x, SurePOS 300/700 or TCxWave family to have the programmable power capability.

In a terminal, the term "powered off" is used to describe various power modes other than full power. If storage retention is enabled, the terminal can enter a standby/suspend mode rather than actually power off. The machine model and BIOS settings can also affect these low power modes.

**Note:** Some terminal types do not support a power-on time but will accept the command to power off or suspend.

This function uses the following parameters:

**FUNC** = 5  
**PARM1** = 2  
**PARM2** = DDHHMMTTT  
**DD** = The day of month (01–31)  
**HH** = The hours (00–24, 99)  
**MM** = The minutes (00–59, 99)  
**TTT** = The terminal number (001–999)

*Disable controller programmable power:* This function is used to disable programmable power on the local controller. This function must be invoked on the controller on which programmable power is to be disabled. The controller that invokes this function must be in the 469x or SurePOS 700 family to have the programmable power capability.

This function uses the following variables:

**FUNC** = 5  
**PARM1** = 3  
**PARM2** = Unused; the value is ignored.

*Enable controller programmable power:* This function is used to enable programmable power on the local controller. This function must be invoked on the controller on which programmable power is to be enabled. The controller that invokes this function must be in the 469x or SurePOS 700 family to have the programmable power capability.

This function uses the following parameters:

**FUNC** = 5  
**PARM1** = 4  
**PARM2** = Unused; the value is ignored.

*Power off a local machine (controller or terminal):* This function is used to power off a local machine. The machine can be a controller or terminal, provided the controller or terminal is in the 469x, SurePOS 300/700 or TCxWave family. This function should be invoked on the machine that is to be powered off. The day, hours, and minutes in the PARM2 string are the power-on time. For terminals, if storage retention is enabled, they will go to a standby or suspend state depending on their hardware configuration. On terminals with storage retention disabled, the terminal will power off and power on at the time entered.

**Note:** Some machines do not support power on time, the time entered is not used so they are only able to power off a terminal or controller. To use the programmable power capability from within an application running locally in the machine, enter the time in the PARM2 parameter.

This function uses the following parameters:

**FUNC** = 5  
**PARM1** = 5  
**PARM2** = DDHHMM  
    **DD** = The day of month (01–31)  
    **HH** = The hours (00–24, 99)  
    **MM** = The minutes (00–59, 99)

*Restart a background application in the same slot:* This function allows a configured background application which has ended to be restarted in the same slot it previously occupied. This service can be used to stop a background application as well.

**Note:** The Name field is not case sensitive but parameters are case sensitive.

Program is responsible for checking user access to this service. This API will only work locally (not remotely from one controller to another controller).

To start a background application, this service uses the following parameters:

**FUNC** = 19  
**PARM1** = 1  
**PARM2** = AAAAAAAAAAAAAAAAAAAAAA[P...]

Where PARM2 is a string of data specifying the application name (1 to 24 characters, indicated by 'A' above) and optionally the parameter string (1 to 45 characters, [P...] above). If there is a parameter string,

the application name must be padded with blanks such that 24 characters are present prior to the parameter data, if needed. The parameter field would then be from 1 to 45 characters.

To stop a background application, this service uses the following parameters:

**FUNC** = 19  
**PARM1** = 2  
**PARM2** = AAAAAAAAAAAAAAAAAAAAAA[P...]

Where PARM2 is a string of data specifying the application name (1 to 24 characters, indicated by 'A' above) and optionally the parameter string (1 to 45 characters, [P...] above). If there is a parameter string, the application name must be padded with blanks such that 24 characters are present prior to the parameter data, if needed. The parameter field would then be from 1 to 45 characters.

**Display application status message:** Interactive applications use this function to display status on the SYSTEM WINDOW CONTROL panel. Applications started in Command Mode cannot use this function.

This function displays the specified text on the WINDOW CONTROL panel. It puts the message in the description field of the window for the application using this function.

The message is available any time you swap to the WINDOW CONTROL panel. It is displayed until the application ends. This message function provides one place where you can look to see the status of all active applications.

This function uses the following parameters:

**FUNC** = 25  
**PARM1** = Unused; the value is ignored.  
**PARM2** = Specify a string containing the message. This should not be modified.

**Display background application status message:** The following function gives status for background applications only. Applications started in Command Mode cannot use this function.

This function displays the specified text on the BACKGROUND APPLICATION CONTROL panel. It puts the message in the message field of the requesting background application. The message is available any time you swap to the BACKGROUND APPLICATION CONTROL panel. It is displayed until the application ends. This message function provides one place where you can look to see the status of all active background applications.

This function uses the following parameters:

**FUNC** = 26  
**PARM1** = Unused; the value is ignored.  
**PARM2** = Specify a string containing the message; this should not be modified.

**Stop application with message:** Interactive applications use this function to display status on the SYSTEM WINDOW CONTROL panel. Applications started in Command Mode cannot use this function. The primary purpose of this function is to handle initialization problems preventing the application from operating. It should not be used once the application has displayed its first panel.

After this function stops the application, it displays the message text that you have specified. It displays this text on the message line of the system screen used to start the requesting application.

This function uses the following parameters:

**FUNC** = 27  
**PARM1** = Unused; the value is ignored.  
**PARM2** = Use to pass a string containing the message; this should not be modified.

The message is displayed only if this function is requested by the current owner of the physical screen.

**Get disk free space:** This function returns the amount of free space on the specified remote or local drive.

The free space is returned in the RET variable. This value defaults to bytes and the maximum value that can be returned is (decimal) 2147483647, which is 2GB – 1. The free space can be returned in kilobytes by including 1024 in the PARM2 selection string variable. This allows the actual free space to be returned when that value is greater than 2GB. For example, if PARM2 is C:\ 1024, the function returns the free space on the C: drive in kilobytes.

This function uses the following parameters:

**FUNC** = 28  
**PARM1** = Unused; the value is ignored.  
**PARM2** = Specify a node name (if non-local) and the drive or a logical name for the node or drive.

**Get configured controllers on the network:** This function returns in the string specified by PARM2 the IDs of all the store controllers on the network. Each ID is two bytes long. Store controller IDs range from CC through ZZ. There can be up to eight controllers configured. If there are fewer than eight, an ID of 00 indicates the end of the list.

This function uses the following parameters:

**FUNC** = 29  
**PARM1** = Unused; the value is ignored.  
**PARM2** = Specify a string variable.

**Get the controller ID for a specified terminal:** This function returns the store controller ID for a terminal you specify.

The data returned is a negative return code, a numeric value for the store controller ID representing the ASCII values CC through ZZ or X'0' if the terminal was not defined.

**Note:** This function returns a controller ID CC through ZZ only if the function was issued at the master store controller or if the terminal is local to the controller issuing the function.

This function uses the following parameters:

**RET** = Always modified.  
**FUNC** = 30  
**PARM1** = Number of the terminal for which the ID is requested.  
**PARM2** = Unused.

**Convert ASCII characters to EBCDIC characters:** This function converts ASCII characters in a string to EBCDIC characters.

This function uses the following parameters:

**FUNC** = 31  
**PARM1** = Unused; the value is ignored.  
**PARM2** = Specify a string of ASCII characters to be converted to EBCDIC; this string is modified to EBCDIC characters.

**Convert EBCDIC characters to ASCII characters:** This function converts EBCDIC characters in a string to ASCII characters.

This function uses the following parameters:

**FUNC** = 32  
**PARM1** = Unused; the value is ignored.  
**PARM2** = Specify a string of EBCDIC characters to be converted to ASCII; this string is modified to ASCII characters.

**Terminal dump preservation:** This function prevents terminal dumps from being written to the terminal dump file until the dump currently in the file can be copied to a diskette. This function is enabled by creating a user logical name ADXTDUMP.

The terminal dump preservation function automatically sets a flag whenever a terminal dump occurs. The preservation flag is reset after the dump is successfully copied to a diskette using the Create Problem Analysis Diskette function. The preservation flag is also reset when the Create Problem Analysis Diskette encounters an incomplete terminal dump and the user chooses to bypass copying the terminal dump to a diskette. While the flag is set, terminal dump requests are rejected so that the dump currently in the terminal dump file is not overwritten.

This function uses the following parameters:

**FUNC** = 33  
**PARM1** = Specify one of the following:  
    **0** To turn off the terminal dump preservation flag  
    **1** To turn on the terminal dump preservation flag  
    **2** To query whether the terminal dump preservation flag is ON or OFF  
**PARM2** = Unused; the value is ignored.

**Get loop message:** This function gets the three most recent system messages for the token-ring adapter specified in PARM2. PARM2 is a string consisting of a node (for example, CD), a TCC Network (1 or 2), and three TCC Network messages.

The node and the TCC Network must be the first three characters of the string when the ADXSERVE function is called. When the ADXSERVE function returns, the three most recent messages follow the node and TCC Network in the string. If no messages exist for the specified node and TCC Network, blanks are returned for the messages. The oldest message is put in the buffer first. Each message is 133 characters long.

This function uses the following parameters:

**FUNC** = 34  
**PARM1** = Unused; the value is ignored.  
**PARM2** = Specify a string containing node and TCC Network.

**Get loop status:**

**Note:** Store Loop support was removed in 4690 OS V6R3.

This function returns the configuration and current status of the two loop adapters. Data is returned in RET in the form WWXXYYZZ (hex) where:

**ZZ** = The first adapter configuration  
**YY** = The first adapter status  
**XX** = The second adapter configuration  
**WW** = The second adapter status

The configuration is defined as:

**00** = Not used  
**01** = Primary  
**02** = Secondary  
**04** = 2400-bps loop  
**80** = Auto-resume of Primary Loop Control

The status is defined as:

**03** = Standby  
**04** = Controlling  
**05** = Prevented

This function uses the following parameters:

**FUNC** = 35  
**PARM1** = Unused; the value is ignored.  
**PARM2** = Unused; the value is ignored.

**Get all active controllers on the network:** This function returns the IDs of all the active store controllers on the network. Each ID is two bytes long. A store controller ID of 00 indicates the end of the list.

This function uses the following parameters:

**FUNC** = 36  
**PARM1** = Unused; the value is ignored.  
**PARM2** = Specify a string variable.

**Get controller ID and loop ID for a specified terminal:** The data returned in RET is two bytes for the Loop ID (1, 2, 3, 4, or 5) followed by the two bytes for the controller ID. A X'01' is returned if the terminal number cannot be found.

**Note:** The RET is valid only if this function is issued at the master store controller.

This function uses the following parameters:

**FUNC** = 37  
**PARM1** = Terminal number.  
**PARM2** = Unused; a string variable.

**Get the controller machine model and type:** This function retrieves the machine model and type and places the 3-byte hexadecimal values to the left in the RET parameter. The 3-byte hexadecimal values are returned in PARM2 if PARM2 is specified. Use the MID\$ statement to extract the individual bytes from PARM2. The MID\$ statement references, from the left, the offset in the PARM2 string.

This function uses the following parameters:

**RET** = INTEGER\*4 with the 3 most-significant bytes containing the machine model and type. The least-significant byte is always zero. Identical to the contents of PARM2.  
**FUNC** = 38  
**PARM1** = Unused, the value is ignored.  
**PARM2** = 3 hexadecimal values left-justified machine model and type. For example, for a 4693-541, the values returned are:  
X'F8'  
X'3E'  
X'00'.

**Check the master or file server exception log:** Use this function to determine whether there are any entries in the exception log.

This function uses the following parameters:

**RET** = One of the following values:  
0 = No entries in the exception log  
1 = Entries exist in the exception log  
8xxxxxxx = Error code indicating why the status of the exception log cannot be obtained.  
**FUNC** = 39  
**PARM1** Specify one of the following values:  
1 = Check the Master exception log  
2 = Check the File Server exception log  
**PARM2** = Unused



**Set terminal sleep mode inactive timeout:** This function enables an application running in a store controller to set the Terminal Sleep Mode Inactive Timeout. When you enable Terminal Storage Retention, you set this value to determine how many minutes a terminal remains idle before going to a suspend or powered down state, depending on the storage retention and the hardware configuration of the terminal.

**Note:** This function is supported only on the store controller. You can also specify the terminal sleep mode inactive timeout by selecting the Enable Terminal Storage Retention option on the TERMINAL FUNCTIONS panel. The default for the terminal sleep mode inactive timeout is 30 minutes.

This function uses the following parameters:

**FUNC** = 41  
**PARM1** = Terminal Sleep Mode Timeout value. Valid values are 0 through 255. You can use a 0 value to disable the terminal sleep mode.  
**PARM2** = Unused; a string variable.

**Enable/disable IPL:** Terminals are able to run offline from the controller. The primary reason to run offline is if there is a problem with the controller.

In many cases, maintenance is applied to the controller to correct the problem. One of the modules that can be applied to the controller as maintenance is the terminal operating system (ADXRT1SL.286). The controller and terminal interact to ensure the terminal is running the latest terminal operating system. The latest is the terminal operating system file currently being used on the controller. The controller is IPLed to apply maintenance. If the terminal is not running the same the terminal is IPLed to load the latest terminal operating system.

When the terminals run offline, the application can write the transaction data to the terminal RAM disk. This enables the terminals to run offline while a controller problem is being addressed. The terminals eventually communicate with the controller and transfer all saved transactions in terminal RAM disk to normal transaction files in the controller. If the terminal is IPLed prior to the completion of the saved transaction movement to the controller, the saved transactions are lost.

Enable/Disable IPL allows an application program that is processing in a terminal to temporarily prevent automatic reloading of the terminal. Automatic terminal reload can occur when the terminal is offline and returns online or when an operator requests the Load Terminal Storage Function with an asterick (\*) specified as the terminal number. Automatic terminal reloads can also occur when ADXCS20L is invoked with the Load All Terminals function code.

Application programs should use the Disable IPL function when the terminal application recognizes that it is offline and begins to save data on the RAM disk. When the terminal application recognizes it is online with the controller, the saved data can be transferred to the controller. After all data has been transferred to the controller, the terminal application can use the Enable IPL function to allow any possible IPL.

The requests to enable and disable the IPL of the terminal for the Mod1 and the Mod2 are handled independently. If a Disable Terminal IPL was outstanding on both terminals and one terminal's application issued an *Enable Terminal IPL* request, an IPL is not possible for the Mod1 and the Mod2 pair. Both terminals must have the IPL enabled before an IPL can occur.

Enable/Disable IPL can also be used to prevent a programmable power command from being executed until the Disable IPL is changed to Enable IPL. At this point, a pending Programmable Power command will be executed.

**Load specific terminal:** This function enables an application running in a store controller to load a specific terminal.

This function uses the following parameters:

**FUNC** = 42



**PARM1** = Terminal Number  
**PARM2** = Unused; a string variable

**Set date and time:** This function enables an application to set the system-wide date and time. Changes made via this function are broadcast to all controllers and all terminals within the store system.

This function uses the following parameters:

**FUNC** = 44  
**PARM1** = Length of string Parm2  
**PARM2** = One of the following string variable commands  
+ = increment the current time by 1 hour  
- = decrement the current time by 1 hour  
B = only broadcast current time to controllers and terminals  
HHMMSS = Hour Minute Second notation for the new current time  
HHMMSS YYMMDD = new Hour Minute Second and Year Month Date

**Note:** Whenever possible, the increment and decrement commands should be used over the set absolute commands because there is no way to guarantee with accuracy when the command will run. This function only runs on the master controller if on a multiple controller system. All of these commands broadcast their changes; you do not need to manually broadcast after making a change.

**Run remote STC in one terminal:** This function causes Remote Set Terminal Characteristics (STC) to be run on one terminal.

The Run Remote STC in One Terminal function uses the following parameters:

**FUNC** = 47  
**PARM1** = Identifies the terminal number that Remote STC is to be run on.  
**PARM2** = Unused, value is ignored.

**Run remote STC in all terminals:** This function causes Remote Set Terminal Characteristics (STC) to be run on in all terminals.

The Run Remote STC in All Terminals function uses the following parameters:

**FUNC** = 48  
**PARM1** = Unused, value is ignored.  
**PARM2** = Unused, value is ignored.

**Using the enable IPL function:** This function enables the terminals to IPL. If terminals were IPLed earlier, but could not because the user had disabled the IPL, requesting this function would cause the terminals to IPL.

To enable terminal IPL, use the following parameters:

**FUNC** = 53  
**PARM1** = Unused; the value is ignored.  
**PARM2** = Unused; the value is ignored.

This function is also used to enable programmable power. It allows the terminal to accept power-off commands. If a previous power-off command had been issued while programmable power had been disabled, and the power-on time had not yet passed, this function would cause the terminal to power down or suspend, depending on if storage retention is enabled and the HW configuration of the terminal.

**Using the disable terminal IPL function:** This function prevents the automatic reload that can occur when a terminal comes online. This function enables the terminal application to effectively use the terminal RAM disk support for temporarily logging data. In most cases, when the controller returns online, the terminal application transfers the saved transaction files to the controller.

To disable terminal IPL, use the following parameters:

**FUNC** = 54  
**PARM1** = Unused; the value is ignored.  
**PARM2** = Unused; the value is ignored.

This function is also used to disable programmable power. It allows the terminal application to block or delay a power down command.

**Wait for system keyboard and mouse inactivity:** This function returns when no controller keyboard, terminal Java keyboard, or mouse activity has occurred on the system console for the time specified (**PARM1**). Refer to “Programming auto sign off” on page 5 for more information about using Auto Sign Off.

This function uses the following parameters:

**FUNC** = 20  
**PARM1** = Number of seconds of keyboard and mouse inactivity to wait for. The allowed range is 15 to 3600 seconds.  
**PARM2** = Unused.

**Wait for system keyboard and mouse activity:** This function returns at the first controller keyboard, terminal Java keyboard, or mouse activity on the system console. Refer to “Programming auto sign off” on page 5 for more information about using Auto Sign Off.

This function uses the following parameters:

**FUNC** = 21  
**PARM1** = Unused.  
**PARM2** = Unused.

**Sign off the system console active user:** This function signs off the system console active user. Refer to “Programming auto sign off” on page 5 for more information about using Auto Sign Off.

This function uses the following parameters:

**FUNC** = 22  
**PARM1** = Unused.  
**PARM2** = Unused.

**Disconnect the system console active user:** This function disconnects the system console active user. Refer to “Programming auto sign off” on page 5 for more information about using Auto Sign Off.

This function uses the following parameters:

**FUNC** = 23  
**PARM1** = Unused  
**PARM2** = Unused

**Start inactivity monitoring:** This function returns when:

- No controller or terminal Java keyboard or mouse activity has occurred on the system console for the time specified, and
- No keyboard activity has occurred on any auxiliary console for the time specified

Refer to “Programming auto sign off for system and auxiliary consoles” on page 6 for more information about using auto signoff for system and auxiliary consoles.

Supported on enhanced systems only.

This function uses the following parameters:

**FUNC**

14

**PARM1**

amount of seconds of keyboard and mouse inactivity to wait for, range 15-3600 seconds (15 seconds – 1 hour)

**PARM2**

Unused

**RET**

= One of the following values:

0x8000 = System console has had no keyboard or mouse activity for the specified amount of time.

0x0001-0x0008 = specified auxiliary console has had no keyboard activity for the specified amount of time

Both the system console and a single auxiliary console can return at the same time. For example, a return value of 0x8003 would indicate that the system console and auxiliary console 3 have had inactivity for the specified amount of time. All consoles that are not specified in the return value will continue to be monitored for inactivity.

**RET**

= One of the following error values:

-1101 = Requestor not a background application.

-1301 = Keyboard and mouse wait already in progress.

-1302 = Amount of seconds to wait for inactivity (PARM1) out of range.

-1304 = No active user on any consoles (system or auxiliary).

**Stop inactivity monitoring:** This function returns when keyboard and mouse inactivity monitoring for all consoles (system and auxiliary) has been stopped.

Supported on Enhanced systems only.

This function uses the following parameters:

**FUNC**

15

**PARM1**

Unused

**PARM2**

Unused

**RET**

= One of the following error values:

-1101 = Requestor not a background application.

-1301 = Keyboard and mouse wait already in progress.

**Signoff specific or all active users:** This function signs off the active user(s) from the requested console(s).

**Supported on enhanced systems only:** This function uses the following parameters:

**FUNC**

16

**PARM1**

Contains the console to signoff the active user.

0 - System console

1-8 = Auxiliary console 1,2,3,4,5,6,7, or 8

-1 = All consoles (system and auxiliary)

**PARM2**

Unused

**RET** = One of the following error values:

- 1101 = Requestor not a background application.
- 1302 = Invalid console requested.
- 1303 = Request not valid for system console when c/t is in terminal mode.
- 1304 = No active user on specified consoles.
- 1305 = Request not valid when there is no auxiliary console configured for this console.

**Disconnect specific or all active users:** This function disconnects the active users from the requested consoles.

Supported on enhanced systems only.

This function uses the following parameters:

**FUNC**

17

**PARM1**

Contains the console to disconnect the active user.

0 - System console

1-8 = Auxiliary console 1,2,3,4,5,6,7, or 8

-1 = All consoles (system and auxiliary)

**PARM2**

Unused

**RET** = One of the following error values:

- 1101 = Requestor not a background application.
- 1302 = Invalid console requested.
- 1303 = Request not valid for system console when c/t is in terminal mode.
- 1304 = No active user on specified consoles.
- 1305 = Request not valid when there is no auxiliary console configured for this console.

**Eject CD/DVD media:** This function allows an application to programmatically eject CD/DVD media inserted in the controller CD/DVD drive.

**Note:** Using the Eject function while writable media is in use may lead to data loss or damage to the media.

In order to prevent data loss, ensure that a *write* operation is not in progress. Issue the **unlockp** command before attempting to eject the media.

New error return codes were added to provide specific results of the Eject command.

This function uses the following parameters:

**FUNC** = 217

**PARM1** = Unused. The value is ignored.

**PARM2** = Unused. The value is ignored.

**RET** = One of the following values:

- 0x805B9700 = A CD/DVD operation was in or is in progress preventing the eject.
- 0x805B9701 = Drive is locked, you must issue an **unlockp** command prior to attempting the eject.
- 0x805B9702 = This is a fatal error and it signals that the low CD/DVD device driver was unable to be opened.
- 0x805B9703 = The unit's door is closing, retry the **eject** command.
- 0x805B9704 = The CD/DVD Device Driver failed to acknowledge the command.

0x805B9707 = The eject was not performed due to a **format** or **chkdsk** command being in progress.

-1020 = A resource on the P: drive is in use, preventing the **eject** command from completing. Usually this indicates that the working directory on the P: drive is set to something other than the root directory.

0 = Success

## Switching between the terminal Java, terminal application, and enhanced mode graphical extensions screens

On a controller/terminal, the system must be on the terminal side for these APIs to display either the terminal Java screen (Java 2 or Java 6, the terminal application screen, or the enhanced mode graphical extensions screen. If a controller screen or command line is being displayed and a terminal application calls this API, you will not be switched away from a controller screen.

**Switch to Java screen:** To function uses the following parameterss:

**FUNC** = 55  
**PARM1** = 1  
**PARM2** Unused; the value is ignored.

This function is used to switch the terminal video so that the terminal Java screen is displayed. It has the same effect as typing **Alt+SysRq+J** on an ANPOS or standard keyboard, or typing S1,5,S2 on the POS keyboard when the terminal application screen is displayed.

Switch to enhanced mode graphical extensions screen

**Switch to terminal screen:** To function uses the following parameterss:

**FUNC** = 55  
**PARM1** = 2  
**PARM2** Unused; the value is ignored.

This function is used to switch the terminal video so that the terminal application screen is displayed. It has the same effect as typing **Alt+SysRq+T** on an ANPOS or standard keyboard, or typing S1,5,S2 on the POS keyboard when the terminal Java screen is displayed.

**Switch to enhanced mode graphical extensions screen:** To function uses the following parameterss:

**FUNC** = 55  
**PARM1** = 4  
**PARM2** Unused; the value is ignored.

This function is used to switch the terminal video so that the terminal application screen is displayed. It has the same effect as typing **Alt+SysRq+X** on an ANPOS or standard keyboard, or typing S1,5,S2 on the POS keyboard. This function is not available on Classic systems.

## Starting a Java application

This function gives the user the ability to start a Java class from BASIC. If the Java Virtual Machine (JVM) starts successfully, the return value is always 0, even if the requested class is not found or even if the process ends in error. To determine if the class started successfully, check your System Message Log. In order to have an interactive mechanism for BASIC to determine if your class started successfully, create a pipe in your Java application. The basic application can query this pipe to check whether it exists.

**Note:** This function is a terminal only function and it is not supported on the controller.

To start a Java application, use the following parameters:

**FUNC** = 56  
**PARM1** = 0  
**PARM2** = A string variable containing the name of the class to be started.

## Stopping a Java application

This function gives the user the ability to stop a Java class from BASIC. The return value is always 0, even if the requested class is not running.

**Note:** This function is a terminal only function and it is not supported on the controller.

To stop a Java application, use the following parameters:

**FUNC** = 57  
**PARM1** = 0  
**PARM2** = A string variable containing the name of the class to be stopped.

## Querying a code page

This function allows the user to query a code page. The function returns a code page number if it is a DBCS locale, otherwise, a negative value is returned.

To query a code page, use the following parameters:

**FUNC** = 58  
**PARM1** = 0  
**PARM2** = Not used.

## Verifying DBCS code points

This function returns 1, if parameter 1 (PARM1) is a valid DBCS character in a current code page. The function returns 0 if PARM1 is not a valid DBCS character.

To verify the DBCS Code Point, use the following parameters:

**FUNC1** = 58  
**PARM1** = Code point  
**PARM2** = Not used.

## ADXERROR (application event logging)

Your application can use a routine called ADXERROR to make entries in the system log and optionally display system messages. The routine should be declared as follows:

```
FUNCTION ADXERROR (TERM,MSGGRP,MSGNUM,SEVERITY,EVENT,UNIQUE) EXTERNAL
INTEGER*2 TERM,MSGNUM,ADXERROR
INTEGER*1 SEVERITY,MSGGRP,EVENT
STRING    UNIQUE
END FUNCTION
```

Invoke the routine using this request:

```
i%=ADXERROR(...)
```

The function always returns zero and uses the following parameters:

**TERM** The parameter is the terminal number. This information comes from the GET APPLICATION STATUS INFORMATION application service request in the terminal. If the application calling this routine resides in the store controller, the TERM parameter should be zero.

**MSGGRP** The (message group) parameter is a one-byte integer that contains the ASCII equivalent of the message group character. The message group is unique for each product and should be in the range of J to S.

**MSGNUM** The message number, if any. The message number should be zero if no message is to be displayed. If MSGNUM is used (non-zero), no Bucket/Source/Event is logged to the file. The system takes this number and converts it to three printable decimal digits, which it appends to the message group to form the message identifier. The message identifier is used to scan the Application Message file. Your application must provide this file for the system message display.

<b>SEVERITY</b>	A number ranging from 1 to 5 that is used to indicate the importance of each message. The system uses the field as a message level indicator. The most important messages have a 1 in this field. The operator can request a message level from 1 to 5, and only messages whose severity number is less than or equal to the current message level appears.
<b>EVENT</b>	A one-byte integer whose value is completely determined by the application. It can be used to indicate why the request is being made. EVENT should be omitted if MSGNUM is used (non-zero value).  For store controller application events 64 to 79, decimal, and terminal application events 64 to 81, decimal, the system uses the log data to generate Network Problem Determination Alert (NPDA) messages. For a description of NPDA support, see the <i>4690 OS: Communications Programming Reference</i> .
<b>UNIQUE</b>	A string of up to 10 bytes of data. This data is included in the log entry made as a result of this request. If the data is longer than 10 bytes, the system uses the first 10 bytes. If it is shorter, the system pads the entry with blanks.

**Note:** The system automatically includes the application name in the log entry.

**Application message logging:** The system references a message display file that your application can provide for messages. This file is called ADXCZOZF.DAT and must be in subdirectory ADX\_IPGM:.

This message file contains fixed length records, so entries must follow this format:

```
cXXX_ssssssss
```

where:

<b>c</b>	= 1-character message group passed through parameter MSGGRP to application services.
<b>XXX</b>	= 3-digit message number passed through parameter MSGNUM to application services.
<b>_</b>	= Blank space
<b>sssssssss</b>	= Message text explaining the message. This text must fill 106 character spaces. If the message text is not that long, pad it with blanks.

**Note:** The 106-character message is displayed as two lines of 53 characters, so be careful of word breaks in your message text.

The system uses the first four characters (cXXX) of the message to scan the message file for a match. If it finds one, the system displays the message identifier and accompanying text on the SYSTEM MESSAGE DISPLAY panel.

---

## Chaining applications

*Chaining* means to transfer control from a currently executing application program directly to another application program. Before the CHAIN statement is issued, any Display Manager files must be closed using the CLSDIS command. Failure to do this results in a loss of system resources needed to open files. Chaining can be performed in one of two ways: chaining to a directly executable module or chaining to an overlay. Directly executable modules have a file extension of 286. Overlays have a file extension of OVR.

See the *4680 BASIC: Language Reference* for more information on chaining.

## Chaining to overlays

Chaining to overlays is supported only in the store controller.

You can share data with an overlay by using common variables. You cannot pass parameters to an overlay with the CHAIN statement. Applications using overlays are made up of a root section and overlay sections.



The root section should define all the public functions and subprograms that overlays need to use. This allows the overlays to share a single definition of the function or subprogram instead of each overlay having its own copy of the function or subprogram. You can chain from the root to an overlay and from one overlay to another. You cannot chain from an overlay to the root.

When you chain to an overlay, execution begins at the first executable statement in the program. If you use overlays, you must link edit your application with its own copy of the runtime subroutine library, because a shared copy of the runtime library does not support overlays.

## Chaining to directly executable modules

Chaining to directly executable modules is supported in both the store controller and the terminal.

You can share data by passing parameters to the modules in the CHAIN statement. When a directly executable module is chained to, execution begins at the first executable statement in the program. Directly executable modules can use a shared copy of the runtime subroutine library.

---

## RAM disk files

The operating system supports a type of in-memory file called *RAM disks*. A RAM disk is a section of memory set aside for use as a high-speed disk storage device. RAM disks improve performance by reducing the amount of time it takes to load frequently-used programs and by redirecting existing application access to files in memory rather than to a hardware disk. RAM disk files can be used for both the store controller and the point-of-sale terminal.

The operating system allows you to optionally install up to four virtual RAM disks in the controller's memory and up to two virtual RAM disks in the terminal's memory. Virtual disks are automatically installed at IPL if your system is configured for RAM disk memory.

The controller's optional disks are named disk devices T:, U:, V:, and W:. Terminal applications can access only disk T:. Controller applications can access all disks.

The terminal's optional disks are named disk devices X: and Y:. Terminal applications can access these disks, but controller applications cannot.

## Considerations for installing RAM disks

RAM disk installation is specified during the configuration process. You can specify the RAM disk size in increments of 32 KB and the number of directory entries in increments of 16.

### Notes:

1. Configuration for RAM disks is done in 32 KB increments, however, the size of the RAM disk created will be rounded to the nearest 64K boundary.
2. Terminal RAM disk is created using the RAM disk information found in the Mod1 terminal device group record only. Mod2 terminal device group records are erased when creating terminal RAM disks.

Some characteristics of RAM disks are fixed and cannot be redefined by the user. These characteristics include:

<b>Sector size</b>	512 bytes
<b>Cluster size</b>	512 bytes
<b>Track size</b>	8 sectors

The maximum disk size is dependent on the terminal memory size, space available, and number of files. Disk memory is allocated in increments of 64 KB. Directory space is allocated in increments of 16 files. FAT space is calculated based on the number of sectors.



**Note:** Configuration for RAM disks is done in 32 KB increments, however the size of the RAM disk created will be rounded to the nearest 64K boundary.

## Accessing RAM disk files from controller applications

In controller applications, all operating system file commands can be executed for RAM disk files, and your 4680 BASIC applications can OPEN, CLOSE, TCLOSE, READ#, or WRITE# to RAM disk files.

Data on the RAM disks is lost during a power outage. Therefore, you should set checkpoints during processing for copying the data from the RAM disk to the hardware disk. The distribution characteristics of RAM disks are similar to those of diskettes: files are not distributed. However, options of ADXCOPYF allow preservation of distribution attributes so that distribution occurs when the files are copied to the hard disk. For information on how to copy files from the RAM disk, refer to the section on “ADXCOPYF (copying RAM disk files)” on page 315.

## Accessing controller RAM disk files from terminal applications

Your terminal application can use all 4680 BASIC statements, which are used for the controller’s hard disks or diskettes, to access RAM disk T: RAM disk U:, V:, and W: cannot be accessed. In addition, use ADXCOPYF to copy files in the controller, regardless of which disk contains them.

## Accessing terminal RAM disk files from terminal applications

Your terminal application can access the controller RAM disk with the same 4680 BASIC statements that it uses to access hard disks and diskettes on the controller. Terminal applications cannot use BASIC statements to access controller RAM disks U:, V:, and W: or terminal hard disk C:. Your application can call ADXCOPYF to copy files from the controller hard disk to the terminal hard disk C:.

### ADXCOPYF (copying RAM disk files)

The subprogram is designed specifically for RAM disk files, but can be used on any files when the target file is on the same controller as the source file. This subprogram is not supported for copying files across the LAN (MCF Network).

**Attention:** If ADXCOPYF fails, the target file is deleted, because partial data might have been written. Therefore, if ADXCOPYF fails while copying to a file that already exists, that file is erased.

When you specify a file name for this routine you must specify the exact file name that you want to copy. You must also specify the exact source and target file names. Do **not** use global (or wildcard) file name characters. The ADXCOPYF subroutine is in the system runtime library: ADXACRCL.L86 for store controller applications, ADXUCRTL.L86 for terminal medium memory model applications, and ADXUCLTL.L86 for terminal big memory model applications.

ADXCOPYF issued from a terminal always attempts to transmit to all terminals requesting a file at the same time. ADXCOPYF uses a broadcast method of sending files from the controller to the terminal, which is the same method used to load terminals. For example, the application .286 file is sent to the terminals using the broadcast method. Using the broadcast method, the operating system on each controller can send only one operating system file and one application file per inter-register communication (i.e., loop1, loop2, LAN, etc.) at a time. Therefore, if half the terminals are using ADXCOPYF to load FILEA.DAT and the other half are trying to load the terminal application .286 file, the second group of terminals must wait until the first group finishes loading FILEA.DAT.

Your 4680 BASIC application should include a declaration similar to the following:

```
SUB ADXCOPYF (RETC,INFILE,OUTFILE,OPT0,OPT1) EXTERNAL
  INTEGER*4 RETC
  STRING    INFILE,OUTFILE
  INTEGER*2 OPT0,OPT1
END SUB
```

The function is invoked as follows:

```
CALL ADXCOPYF (retc,infile,outfile,opt0,opt1)
```

Where:

**retc** = A 4-byte integer return code.

**infile** = A string containing the file name to be copied from (source file).

**Note:** For the terminal ADXCOPYF, the total length of a controller input file name must be less than 25 characters. You can use a logical name to avoid this restriction. See “Using logical names” on page 20 for more information.

**outfile** = A string containing the file name to be copied to (target file).

**opt0** = A 2-byte integer indicating whether or not copy operation should be performed:

**0** = File is copied whether or not the output file already exists.

**1** = Error message (-2008) is returned if the file already exists and the file is not copied.

**opt1** = A 2-byte integer used to indicate distribution on a LAN (MCF Network) system (only for 4680 Version 2):

**0** = Keep the distribution on the output file the same as on the input file.

**1** = If the output file exists, keep the same distribution.

**2** = Make the output file local.

Table 34 on page 316 shows the ADXCOPYF defined return codes and their descriptions:

Table 34. ADXCOPYF return codes

Return code	Description
0	Successful copy
Error codes for open on input file	
-2001	An incorrect file name on the input file
-2002	File in use on the input file
-2003	Input file not found
-2004	Ownership differences
-2005	Incorrect device for input file
Error codes for open on output file	
-2006	Invalid file name on the output file
-2007	File in use on the output file
-2008	File already exists and user-specified not to delete
-2009	Incorrect device for output file
Error codes for reading the input file	
-2011	Error reading the input file
Error codes for writing to the output file	
-2016	Hardware error writing to the output file
-2017	Insufficient space for the output file
-2018	Unable to allocate Read/Write buffer for copy
Error code for closing the input file	
-2021	Error closing the input file
Error code for closing the output file	
-2026	Error closing the output file

Table 34. ADXCOPYF return codes (continued)

Return code	Description
Error code for renaming the temporary file	
-2031	Rename error

## Store controller application services

This section contains routines that are available only to applications on the store controller. These routines are in runtime library ADXACRCL.L86.

### ADXFILES (canceling the shared use of a file)

ADXFILES routines cancel the shared use of a file such as the transaction log. Without the use of ADXFILES, a shared file cannot be renamed or deleted because it is in use by more than one user.

Some of the conditions that can cause file sharing problems are incomplete transactions, terminal hardware failures, incomplete controller functions, and software failures. ADXFILES provides these functions to manage the canceling of shared file usage:

- Restricting a file for exclusive use
- Unrestricting a file that was restricted
- Determining despool status

Your 4680 BASIC application should include a declaration similar to the following:

```
SUB ADXFILES (RET, FUNC, PARM1, PARM2) EXTERNAL
INTEGER*4 RET
INTEGER*2 FUNC, PARM1
STRING    PARM2
END SUB
```

### ADXFILES restrict

ADXFILES restrict forces the exclusive use of a single file. This function is intended to be used only for store closing procedures and should not be used as a general purpose function.

ADXFILES restrict function causes all applications currently using that file to lose their access to that file until they have closed and reopened the file. When an application attempts to use a file that has been restricted, the file request is rejected and a new return code is returned. In the controller the return code is 80F306F0 and the associated ERR code in BASIC is \*I. In the terminal, the first terminal access to a restricted file receives a 80F306F0 and subsequent accesses (by the same or other terminals) receive a 80004007 (bad file number). Both of these return codes in the terminal have an associated ERR code of \*I. When an application has a file open and receives either of these return codes for a file request, it should close and reopen that file.

- Purpose:
 

To restrict the use of a file by all other applications. The restrict applies to applications on all controllers and all terminals. To restrict the use of a mirrored or compound file you restrict the prime version of the file on the file server or master, respectively. You cannot use restrict for a distribute-on-close type file. After a file is restricted:

  - It cannot be opened by any application.
  - An application that is using a file that is restricted by another application must close and reopen a file to use the file again.
- Restrictions:
  - Only one ADXFILES to restrict a file can be active at a time.
  - To restrict a file on a file server or a master, the controller application executing the restrict must be connected to the active file server or master.
  - To restrict a mirrored file or compound file, an application must restrict the prime version of the file.

- A distribute-on-close type file cannot be restricted.
- Location of usage:
  - Any application on any controller.
  - It cannot be used by a terminal application.

- Calling sequence:

CALL ADXFILES (*ret,func,parm1,parm2*)

where:

**RET** = 4-byte integer that indicates the status of the restrict function

**hi lo**

**xx xx yy yy** = Indicates how the file was being used when the restrict was executed:

**xxxx** = Number of other controller applications that were using this file when it was restricted.

**yyyy** = Number of controllers where terminal applications were using this file when it was restricted.

**0000** = No other application has the file open.

**80 F3 06 F4** = Application attempted to restrict a file while another restrict is active.

**80 F3 06 F5** = Application attempted to restrict a distribute-on-close file.

**80 F3 06 F6** = Application attempted to restrict the file on the wrong node.

**8x xx xx xx** = Any other operating system error return code.

**-1201** = Invalid function code used for the FUNC values defined for ADXFILES.

**-1202** = PARM1 is not defined.

**-1203** = PARM2 is not defined.

**-1204** = PARM2 is not a valid file name.

**-1205** = Force Close support is not installed.

**FUNC** = 1 (This is the restrict function code for ADXFILES.)

**PARM1**

= Unused; the value is ignored.

**PARM2**

= String containing the file name of the file to be restricted. The name can be a logical name or a fully-qualified file name. To reference a mirrored or compound file use the generic node names for the file server and the master:

- for file server use ADXLXACN::
- for master use ADXLXAAN::

**Note:** When an application attempts to use a file that has been restricted, you receive either an 80F306F0 or an 80004007 (in BASIC, ERR code = \*I). The error recovery should not be done in the ON ERROR code. The ON ERROR code should indicate that an error has occurred and the file should be closed and reopened. The indication should be made to the code that uses the file. The reopen should provide a delay and retry mechanism because the file remains restricted until it is unrestricted by the application.

For example, you can declare a global variable in the program. If the ERR code is \*I and ERRN is either 80F306F0 or 80004007, set a special value to the global variable in the ON ERROR routine to indicate that the file has been restricted by another application. Then RESUME to the statement following the failing statement. The program can then check for the special value in the global variable set in the ON ERROR routine. If the value is set, perform the error recovery.

## ADXFILES unrestrict

ADXFILES unrestrict stops the effect of the ADXFILES restrict. The unrestrict is requested for the same file name that was used for the restrict even if the file was renamed while it was restricted. After the unrestrict is executed, that file name can be used by all applications again.

- Purpose:

To unrestrict the use of a file that had been previously restricted. To unrestrict the use of a mirrored or compound file, unrestrict the prime version of the file on the file server or master, respectively. After a file is unrestricted:

- The file name can be used to open files according to the normal guidelines.
- An application that lost access to a file due to restrict must issue a CLOSE followed by an OPEN after the unrestrict is issued to gain access to the file again.

- Prerequisites:

The application executing the unrestrict ADXFILES must have executed the restrict ADXFILES.

- Restrictions:

If an application is unrestricting a file on a file server or a master, the controller executing the application must be connected to the active file server or master.

- Location of usage:

- Any application on any controller.
- It cannot be used by a terminal application.

- Calling sequence:

CALL ADXFILES (RET, FUNC, PARM1, PARM2)

**RET** = 4-byte integer that identifies the unrestrict status.

**hi lo**

**00 00 00 00** = Unrestrict is successful.

**80 F3 06 F2** = Application attempted to do an unrestrict without doing a restrict.

**80 F3 06 F3** = Application attempted to unrestrict a file that it did not have restricted.

**8x xx xx xx** = Any other OS error return code.

**-1201** = Invalid function code.

**-1202** = PARM1 is not defined.

**-1203** = PARM2 is not defined.

**-1204** = PARM2 is not a valid file name.

**-1205** = Force Close support is not installed.

When an unsuccessful unrestrict is executed because of a LAN failure (80 60 xx xx), the application should wait 30 seconds and retry the unrestrict. This should be repeated for a total of 7 executions of unrestrict.

**FUNC** = 2 (This is the unrestrict function code for ADXFILES.)

**PARM1** = Unused; the value is ignored.

**PARM2** = String containing the file name of the file to be restricted. The name can be a logical name or a fully qualified file name. To reference a mirrored or compound file use the generic node names for the file server and the master store controller:

- For file server use ADXLXACN::
- For master use ADXLXAAN::

## ADXFILES despool

ADXFILES despool determines how many total bytes of data remain in all the spool files and how many controllers have spool files. By using this function, applications can determine that the store personnel should be notified that a store closing must be delayed and can give the store personnel an idea of the length of the delay.

- Purpose:

This function determines the status of the operating system despooling of files. This function determines how many bytes of data are yet to be despoiled and how many controllers have data in their spool files to be despoiled. The values determined by this function are probably different than the sizes of the spool files because the size of the spool files are not set to zero until all the data has been despoiled from them.

- Prerequisites:

None currently identified.

- Restrictions:

This function can only determine despool status for controllers that are currently in session with this controller on the LAN.

- Location of Usage:

- Any application on any controller.
- It cannot be used by a terminal application.

- Calling sequence:

CALL ADXFILES (RET, FUNC, PARM1, PARM2)

**RET** = 4-byte integer that specifies the despool status.

**hi lo**

**0w xx yy yy** = where:

**w** = Status of LAN communications.

**0** = Means all controllers are communicating with this controller.

**8** = Means one or more controllers are not communicating with this controller.

**xx** = Number of communicating controllers with data to despool.

**yyyy** = Total number of bytes (in 1000s ) of data to be despoiled by controllers communicating.

**00 00 00 00** = The system supports LAN and there is no despooling to do or the system does not support LAN.

**8x xx xx xx** = Any operating system error return code.

**-1201** = Invalid function code.

**-1202** = PARM1 is not defined.

**-1205** = System supports LAN but Force Close support is not installed.

**FUNC** = 3 (This is the despool function code for ADXFILES.)

**PARM1**

= Unused; the value is ignored.

**PARM2**

= Unused; the value is ignored.

## ADXCSE0L (store closed query application)

ADXCSE0L is a operating system application that determines if the store has been closed to allow end of day processing to begin.

For example, you might want to start the end of day processing in every store in your store system network. Before starting this processing you must be sure that each store is closed. *Closed* means you are no longer processing transactions. To determine if all the stores are closed, invoke ADXCSE0L in each store using the NetView DM INITIATECLIST command. The code returned to NetView® DM from the controller can indicate the store is still processing sales transactions. In this case the NetView DM plan that starts the end of day processing bypasses the store.

The ADXCSE0L application is written in 4680 BASIC and allows for a user exit. The user exit determines the status of the store. If a zero (0) is returned to the main program, the store is closed. If the store is open, the return code is a negative one (-1). The return value is defined as INTEGER\*4. ADXCSE0L returns the code to the operating system.

A user exit is required because determination of the store being closed differs for each sales application product. The name of the user exit must be ADXCSEUR. You must link your user exit with the base application to form ADXCSE0L. The default user exit returns a store closed status of -1, which is open.

The purpose of this user exit routine is to enable you to write code in 4680 BASIC to accomplish any processing that you want. For example, printing reports or saving critical data files.

The subroutine should terminate with a recognizable return code that is returned to the host.

A user exit subroutine is available for the General Sales Application. It queries the store status field in the terminal status file and return a code of zero (closed) or negative one (open). For other sales applications, a user exit is not available.

## ADXEXIT (set the ERRORLEVEL VALUE)

ADXEXIT can be used in a BASIC application to set the ERRORLEVEL value that can be tested in a .BAT file. This value provides a way for a BAT program to test the results of a BASIC program that it has invoked.

Calling ADXEXIT terminates the calling program and sets a 2-byte integer that is used to set the ERRORLEVEL value for a .BAT file. To call ADXEXIT in a BASIC program, you must first define it as follows:

```
SUB ADXEXIT (PARM1) EXTERNAL
INTEGER*2 PARM1
END SUB
```

To call ADXEXIT, your code should be similar to the following:

```
ERRORLEVEL = 20
CALL ADXEXIT (ERRORLEVEL)
```

The allowed values for the 2-byte integer are 0 to 32,767.

In addition, the BASIC program calling ADXEXIT must be linked with the ADXACRCL.L86 file containing ADXEXIT.

To test the value of errorlevel in a .BAT file, test in descending order as in the following example:

```
IF ERRORLEVEL 500 GOTO T500
IF ERRORLEVEL 20  GOTO T20
IF ERRORLEVEL 19  GOTO T19
IF ERRORLEVEL 1   GOTO T1
    (this statement would be executed if ERRORLEVEL is 0)
:T500
    (this statement would be executed if ERRORLEVEL is 500 or more)
:T20
    (this statement would be executed if ERRORLEVEL is 20 to 499)
:T19
    (this statement would be executed if ERRORLEVEL is 19)
:T1
    (this statement would be executed if ERRORLEVEL is 1 to 18)
```

This section lists functions available only to applications running at the store controller. Use the ADXSERVE request call for these functions.

## ADXSERCL (closing an application service)

This subprogram is used by controller applications to close ADXSERVE. The ADXSERCL subprogram is to be used only before chaining from an application that invoked ADXSERVE. If ADXSERVE was not invoked, ADXSERCL does not affect the operating system.

ADXSERCL allows applications that chain to deallocate system resources. These resources are allocated the first time the application uses ADXSERVE. It is not necessary to close ADXSERVE each time it is used. If an ADXSERCL is used after each execution of ADXSERVE, system performance is impaired.

**Attention:** If you have invoked ADXSERVE and you are not chaining applications, ADXSERCL must not be used.

The ADXSERCL routine for store controller applications is written in 4680 BASIC. The application should include the following declaration statement:



```
SUB ADXSERCL EXTERNAL
END SUB
```

The subprogram is invoked as follows:

```
Call ADXSERCL
```

---

## Terminal application services

This section lists routines that are available only to applications at the terminal.

### ADXDIR (listing terminal RAM disk files)

Terminal applications use the ADXDIR subprogram to list the files in RAM disks X: and Y: and hard disk C:. ADXDIR also shows the amount of free space on the disks. This subprogram displays the same type of information about terminal RAM disks that the DIR command displays about controller disks. The ADXDIR subprogram is in the system runtime subroutine library ADXUCRTL.L86 for medium memory model terminal applications and ADXUCLTL.L86 for big memory model terminal applications.

The 4680 BASIC terminal application must contain a procedure definition similar to the following:

```
SUB ADXDIR (RETC,DISKID,OUTINFO,FILEINDX) EXTERNAL
INTEGER*4 RETC
STRING DISKID,OUTINFO
INTEGER*2 FILEINDX
END SUB
```

The SUBPROGRAM is called as follows:

CALL ADXDIR (*retcode,disk,direntry,opt*)

where:

- retcode** = 4-byte integer return code.
- disk** = String containing the disk ID (X:, Y:, or C:) and optionally continuing with a file name with or without global file name characters (\*). One level of subdirectory is supported only for the hard disk C:.
- direntry** = String that contains information for one directory entry on return to the caller.
- opt** = Option indicating whether the first or next directory entry is wanted. An integer value of 0 or 2 indicates first, and 1 or 3 indicates next.  
  
This option also controls the format of the data returned to the caller. Options 3 and 4 permit larger values to be returned for free space, number of files, and file size as documented below.

The ADXDIR subprogram is typically called within a loop in the application, such that each repetition of the loop passes the information for a directory entry. The *opt* parameter should indicate the first entry on the first repetition of the loop, and it should indicate the next entry on the following repetition. When the application is executing such a loop, the ADXDIR subprogram passes information for each file on the RAM disk or the hard disk in the *direntry* string as the loop progresses. As the loop is executed and file information is passed in the *direntry* string, the application can display it, collect it, or send it to an application in the controller, for example. When information for all existing files has been passed, the *retcode* indicates this condition and the application can terminate the loop.

Table 35. Successful ADXDIR return codes that do not imply error conditions.

Return code	Description
0	Indicates successful passing of file information
1	Indicates no more files on the disk



Table 36. ADXDIR error return codes that imply error conditions.

Return code	Description
-3001	Indicates the disk ID is not X:, Y:, or C:
-3002	Indicates the requested X:, Y:, or C: disk is not configured.

The *direntry* string is formatted as follows on return from ADXDIR:

### Options 1 and 2:

If the return code is 0:

Table 37. Return code 0 string formatting

	Offset	Field length
file name	0	8 characters
file name extension	9	1 character
file size	13	3 characters
date last modified	21	1 character
time last modified	30	7 characters
disk free space	41	7 characters

If the return code is 1, the string contains the number of files and the bytes of free space, and is formatted as follows. (Unused fields contain blanks.)

Table 38. Return code 1 string format

	Offset	Field length
number of files	37	3 characters
disk free space	41	7 characters

### Options 2 and 3:

These options perform like options 0 and 1, but allow for larger values to be returned for file size, number of files, and free space. If the return code is 0, the string is formatted as follows:

Table 39. Return code 0 string format

	Offset	Field length
file name	0	8 characters
file name extension	9	3 characters
file size	13	10 characters
date last modified	24	8 characters
time last modified	33	6 characters
disk free space	46	10 characters

If the return code is 1, the *direntry* string contains the number of files and the bytes of free space, and is formatted as follows.

Table 40. Return code 1 string format

	Offset	Field length
number of files	40	5 characters
disk free space	46	10 characters

File size, free space, and number of files are right justified in their fields. Fields are separated by blanks.

## Extended memory management for the Point of Sale terminal

Extended memory management is an alternative to terminal RAM disk. It is typically used when the terminal does not have enough memory for the RAM disk, but the application needs more memory for data than its 64-KB data segment. Extended memory management is a library of subroutines that is linked with the terminal application. There are two sets of libraries available that contain these subroutines: ADXMEM0L.L86 and ADXMEM1L.L86 are used when linking an application that was compiled using the 4680 CBASIC medium memory model; ADXMEL0L.L86 and ADXMEL1L.L86 are used when linking an application that was compiled using the 4680/90 CBASIC big memory model.

ADXMEM0L.L86 and ADXMEL0L.L86 libraries contain subroutines that enable the application to allocate, free, read, write, and search the memory. ADXMEM1L.L86 and ADXMEL1L.L86 libraries contain subroutines that enable the application to insert and delete keyed records in memory. Applications in the Mod1 and Mod2 terminals can share a section of memory while using any of these routines.

The subroutines are linked by adding the appropriate library to your link input file.

### Notes:

1. If you are a *medium* memory model user: If you use only extended memory management routines contained in the ADXMEM0L.L86 library, you do not need to link with the ADXMEM1L.L86 library. If you use any routines contained in the ADXMEM1L.L86 library, you must also link with the ADXMEM0L.L86 library. Also, the ADXMEM1L.L86 library must precede the ADXMEM0L.L86 library in the link order to avoid link errors.
2. If you are a *big* memory model user: If you use only extended memory management routines contained in the ADXMEL0L.L86 library, you do not need to link with the ADXMEL1L.L86 library. If you use any routines contained in the ADXMEL1L.L86 library, you must also link with the ADXMEL0L.L86 library. Also, the ADXMEL1L.L86 library must precede the ADXMEL0L.L86 library in the link order to avoid link errors.

The following table describes each subroutine and defines its library name.

Subroutine	Description	Medium memory model library name	Big memory model library name
GETMEM	Allocate a memory file	ADXMEM0L.L86	ADXMEL0L.L86
OPENMEM	Gain access to a shared memory file	ADXMEM0L.L86	ADXMEL0L.L86
MEMSYN	Gain mutually exclusive access to shared memory	ADXMEM0L.L86	ADXMEL0L.L86
MEMUNSYN	Release mutually exclusive access to shared memory	ADXMEM0L.L86	ADXMEL0L.L86
FREEMEM	Free a memory file	ADXMEM0L.L86	ADXMEL0L.L86
AVAILMEM	Query available free memory	ADXMEM0L.L86	ADXMEL0L.L86
MEMWRITE	Write data to a memory file	ADXMEM0L.L86	ADXMEL0L.L86
MEMREAD	Read data from a memory file	ADXMEM0L.L86	ADXMEL0L.L86

Subroutine	Description	Medium memory model library name	Big memory model library name
MEMSRCHB	Binary search for matching field	ADXMEM0L.L86	ADXMEL0L.L86
MEMSRCHS	Sequential search for matching field	ADXMEM0L.L86	ADXMEL0L.L86
MEMWRKEY	Insert keyed records into a memory file sequenced in ascending order by key	ADXMEM1L.L86	ADXMEL1L.L86
MEMDLKEY	Delete keyed records from a memory file sequenced in ascending order by key	ADXMEM1L.L86	ADXMEL1L.L86
MEMCLEAR	Clear a memory file	ADXMEM1L.L86	ADXMEL1L.L86

## Using extended memory management

A section of memory that is managed by these subroutines is called an *in-memory file*. An in-memory file is not the same as a RAM disk file. In-memory files have much less overhead than RAM disk files. In-memory files are intended for terminals that have 1 MB of memory. Normally these terminals do not contain enough memory for RAM disk use.

The subroutines can be divided into five different categories, according to their functions.

1. The first category contains subroutines that are used to access or to release access to memory. GETMEM is used to allocate a section of memory of a size specified by the application. GETMEM has an option to allocate a section of memory that can be shared by other applications. If another application needs to access an in-memory file that was allocated in this way, it issues an OPENMEM call. It can then access the in-memory file just as if it had allocated it by issuing GETMEM. FREEMEM is used to release access to an in-memory file. In the case of shared in-memory files, the memory is released when the last application issues the FREEMEM against the in-memory file.
2. The second category contains subroutines that read and write the memory. These subroutines function the same whether or not the in-memory file is shared. The simplest subroutines are MEMREAD and MEMWRITE.  
  
MEMSRCHS is a routine that is used for a sequential search of an in-memory file for a particular key. MEMSRCHB is used for a binary search. MEMSRCHB should be used for doing searches only if the in-memory file is sequenced by a key in ascending order; otherwise MEMSRCHS should be used. The MEMWRKEY subroutine is used to build a key sequenced memory file. A binary search is faster than a sequential search, but a binary search does not work if the data is not sorted.  
  
Using the MEMSRCHS and MEMSRCHB subroutines, the caller passes a search argument, search argument length, and the offset into the in-memory file records of the key. The search argument is compared with the key. If they match, the search completes successfully.
3. The third category of subroutines is used by applications to synchronize access to shared in-memory files. If an application temporarily needs exclusive access to a shared memory file, it should call MEMSYN before beginning the access. After it is finished with the access it should call MEMUNSYN. Following is an explanation of how MEMSYN and MEMUNSYN are used. There are two applications: application A and application B. They share an in-memory file. The in-memory file contains a counter that is incremented at the end of every transaction, so that it contains the total number of transactions that have occurred on the Mod1 and Mod2 pair. At the end of each transaction, the application must read in the counter using MEMREAD, increment it, and write it back using MEMWRITE.  
  
If the applications are not using MEMSYN and MEMUNSYN, the following happens: application A finishes a transaction. It reads in the counter using MEMREAD. Application A is preempted by application B, which is also finish a transaction and read in the counter. Application B increments the counter and writes it back using MEMWRITE. Application A runs again. It increments its copy of the counter and write it back. At that point, the counter is less than it should be because the applications

are not guaranteed exclusive access. However, if each of the applications had called MEMSYN before calling MEMREAD; and called MEMUNSYN after calling MEMWRITE, this problem would not have happened.

If an application calls MEMSYN and then another application calls MEMSYN using the same in-memory file number, execution of the second application is suspended until the first application calls MEMUNSYN.

**Note:** When a shared in-memory file is created, the GETMEM subroutine performs a MEMSYN. Until the application that performed the GETMEM calls MEMUNSYN, the execution of any application performing a MEMSYN is suspended.

4. The fourth category is the subroutine AVAILMEM. AVAILMEM is called to find out how much free memory is in the system.

Each of the calls (excluding AVAILMEM) provides an in-memory file number. When the GETMEM or the OPENMEM is performed, the subroutines associate that in-memory file number with a section of memory until the FREEMEM is done.

One application cannot have more than one in-memory file with the same in-memory file number. If two applications use the same in-memory file number for non-shared in-memory files, the in-memory file number refers to a different section of memory for each application.

If an application allocates a shared in-memory file, the other application accesses that shared in-memory file by issuing an OPENMEM with the same in-memory file number as was used by the application that called GETMEM.

5. The fifth category contains the subroutines, MEMWRKEY and MEMDLKEY. MEMWRKEY and MEMDLKEY are used to insert and delete keyed records in an in-memory file. MEMWRKEY and MEMDLKEY do not work if the in-memory file is not sorted in ascending order by key. Creating a file using MEMWRKEY automatically sorts the in-memory file in ascending order by key.

Using MEMWRKEY, the in-memory file is searched for a record. If a record with the same key is found, MEMWRKEY overlays the old record. If the record that was found does not have the same key, then the MEMWRKEY creates space for the new record by moving up each record with a greater key. An argument is passed to MEMWRKEY that indicates the length of the key, and the size of the data including the key.

Using MEMDLKEY, the in-memory file is searched for a record. If the record is found the keyed record is deleted and all records with a greater key are shifted down by one record. If MEMDLKEY does not find the specified record, an error message indicating the record was not found is returned to the application.

MEMCLEAR can be used to delete all records from the in-memory file. The entire memory space is filled with X'0xFF' and the internal record counter is reset to zero.

**Note:** When using MEMWRKEY and MEMDLKEY, the key must start with the first byte of the record. If MEMWRKEY, MEMDLKEY, or MEMCLEAR are being used on a shared in-memory file, the application should call MEMSYN before calling MEMWRKEY, MEMDLKEY, or MEMCLEAR followed by calling MEMUNSYN.

## Defining extended memory management subroutines

The following section contains an explanation of the subroutines that are supported by extended memory management.

**Note:** The following explains the parentheses within the parameter list:

- *I* passes input data to the memory manager subroutine.
- *O* passes output data from the memory manager subroutine.
- *I/O* passes input data to and output data from the memory manager subroutine in the same parameter.

**Attention:** Before reading data from a memory file, a string of data must be allocated in 4680 BASIC. The memory manager does not allocate or increase the size of the receiving string.

The subroutines by entry point are:

## GETMEM

Allocates a memory file

- Call Parameters:
  - (O) INTEGER\*4 return value receive area
  - (O) INTEGER\*4 record count or system error receive area
  - (I) INTEGER\*2 memory file number
  - (I) INTEGER\*4 count of memory records to allocate
  - (I) INTEGER\*2 memory record size
  - (I) INTEGER\*2 shared memory flag
    - 0 = not shared
    - 1 = shared
- Return Value:
  - Zero if file is successfully created
  - Negative return code on error

### Notes:

1. GETMEM performs a MEMSYN when creating a shared in-memory file. MEMUNSYN should be called after GETMEM to enable the next function that issues a MEMSYN to obtain exclusive access to the in-memory file. If MEMUNSYN is not called after GETMEM, the next function to issue a MEMSYN is suspended until a MEMUNSYN is issued.
2. The memory file number is set by the 4680 BASIC program. The size of the in-memory file that is allocated is obtained by multiplying the record count by the record size. Shared in-memory files are limited in size to 64 Kb minus 12 bytes. A record count is returned to the caller to indicate the number of records that were allocated. Non-shared files can be as large as the available memory in the terminal.

## OPENMEM

Gains access to a shared memory file

- Call Parameters:
  - (O) INTEGER\*4 return value receive area
  - (O) INTEGER\*4 record count or system error receive area
  - (I) INTEGER\*2 memory file number
  - (I) INTEGER\*2 memory file record size
- Return Value:
  - Zero if file is successfully created
  - Negative return code on error

**Note:** The record size specified on the OPENMEM should be the same as the record size specified on the GETMEM. However, no automatic checking is done for this. A record count that is returned to the caller indicates the size of the in-memory file.

## MEMSYN

Gains mutually exclusive access to shared memory

- Call Parameters:
  - (O) INTEGER\*4 return value
  - (O) INTEGER\*4 system error receive area
  - (I) INTEGER\*2 memory file number
- Return Value:
  - Zero if file is successfully created
  - Negative return code on error

**Note:** An error results if this call is issued using a file number for a non-shared in-memory file.

## MEMUNSYN

Releases mutually exclusive access to shared memory

- Call Parameters:
  - (O) INTEGER\*4 return value
  - (O) INTEGER\*4 system error receive area
  - (I) INTEGER\*2 memory file number
- Return Value:
  - Zero if file is successfully created
  - Negative return code on error

**Note:** An error results if this call is issued using a file number for a non-shared in-memory file.

## FREEMEM

Frees a memory file

- Call Parameters:
  - (O) INTEGER\*4 return value receiving area
  - (O) INTEGER\*4 system error code receiving area
  - (I) INTEGER\*2 memory file number
- Return Value:
  - Zero if successful
  - Negative return code on error

## AVAILMEM

Queries available free memory

- Call Parameters:
  - (O) INTEGER\*4 return value receiving area
  - (O) INTEGER\*4 system error code receiving area
- Return Value:
  - Amount of free memory available in bytes

## MEMWRITE

Writes data to a memory file

- Call Parameters:
  - (O) INTEGER\*4 return value receiving area
  - (O) INTEGER\*4 system error code receiving area
  - (I) INTEGER\*2 memory file number
  - (I) STRING data to be written
  - (I) INTEGER\*4 target memory record number
  - (I) INTEGER\*2 offset into target record
  - (I) INTEGER\*2 length of data (0 defaults to record size)
- Return Value:
  - Zero if successful
  - Negative on error

## MEMREAD

Reads data from a memory file

- Call Parameters:
  - (O) INTEGER\*4 return value receiving area
  - (O) INTEGER\*4 system error code receiving area
  - (I) INTEGER\*2 memory file number
  - (O) STRING receiving string for the data
  - (I) INTEGER\*4 target memory record number
  - (I) INTEGER\*2 offset into target record
  - (I) INTEGER\*2 length of data (0 defaults to record size)
- Return value:

- Zero if successful
- Negative on error

**Note:** Before calling MEMREAD, allocate a string that is large enough to receive the data.

## MEMSRCHB

Conducts binary search for matching field

- Call Parameters:
    - (O) INTEGER\*4 return value receiving area
    - (O) INTEGER\*4 system error code receiving area
    - (I) INTEGER\*2 memory file number
    - (I/O) STRING search argument
    - (I) INTEGER\*2 length of search argument
    - (I) INTEGER\*2 offset into target record of field, offset of 0 is valid
    - (I) INTEGER\*2 length of return data

If 0, no data is returned. If greater than 0, data is returned to the search argument.

  - (I) INTEGER\*4 offset of return data field in record, offset of 0 is valid
- Return value:
    - Zero if successful
    - Negative on error

**Note:** The file must be sorted in ascending order to use the binary search. If the file is partially filled with records, it should be initialized with records of hexadecimal X'FF' to ensure ascending order. If data is to be returned, allocate the argument string large enough to receive the data.

## MEMSRCHS

Performs sequential search for matching field

- Call Parameters:
    - (O) INTEGER\*4 return value receiving area
    - (O) INTEGER\*4 system error code receiving area
    - (I) INTEGER\*2 memory file number
    - (I/O) STRING search argument
    - (I) INTEGER\*2 length of search argument
    - (I) INTEGER\*2 offset into target record of field, offset of 0 is valid
    - (I) INTEGER\*2 length of return data

If 0, no data is returned. If greater than 0, data is returned to the search argument.

  - (I) INTEGER\*4 offset of return data field in record, offset of 0 is valid
- Return value:
    - Zero if successful
    - Negative on error

**Note:** If data is to be returned, allocate the argument string large enough to receive the data.

## MEMWRKEY

Inserts data to a memory file in ascending order by key

- Call parameters:
  - (O) INTEGER\*4 return value receiving area
  - (O) INTEGER\*4 system error code receiving area
  - (I) INTEGER\*2 memory file number
  - (I) STRING data to be written
  - (I) INTEGER\*2 length of key
  - (I) INTEGER\*2 length of data (0 defaults to record size)
- Return value:
  - Zero if successful



- Negative on error

**Notes:**

1. When using MEMWRKEY, the key must start with the first byte of the record.
2. If MEMWRKEY is being used on a shared in-memory file, the application should call MEMSYN before calling MEMWRKEY followed by calling MEMUNSYN.

**MEMDLKEY**

Deletes data from a memory file sequenced in ascending order by key

- Call Parameters:
  - (O) INTEGER\*4 return value receiving area
  - (O) INTEGER\*4 system error code receiving area
  - (I) INTEGER\*2 memory file number
  - (I) STRING key of record to be deleted
  - (I) INTEGER\*2 length of key
- Return Value:
  - Zero if successful
  - Negative on error

**Notes:**

1. When using MEMDLKEY, the key must start with the first byte of the record.
2. If MEMDLKEY is being used on a shared in-memory file, the application should call MEMSYN before calling MEMDLKEY followed by calling MEMUNSYN.

**MEMCLEAR**

Clears a memory file

- Call parameters:
  - (O) INTEGER\*4 return value receiving area
  - (O) INTEGER\*4 system error code receiving area
  - (I) INTEGER\*2 memory file number

**Note:** If MEMCLEAR is being used on a shared in-memory file, the application should call MEMSYN before calling MEMCLEAR followed by calling MEMUNSYN.

**Example subroutine declarations using 4680 BASIC**

The following examples explain how to format a 4680 BASIC declaration.

```
SUB GETMEM (RETCODE,SYSRET,FILEID,KOUNT,RECSIZE,SHFLAG) EXTERNAL
  INTEGER*4 RETCODE, SYSRET, KOUNT
  INTEGER*2 FILEID, RECSIZE, SHFLAG
END SUB
```

```
SUB OPENMEM (RETCODE,SYSRET,FILEID,RECSIZE) EXTERNAL
  INTEGER*4 RETCODE, SYSRET
  INTEGER*2 FILEID, RECSIZE
END SUB
```

```
SUB MEMSYN (RETCODE,SYSRET,FILEID) EXTERNAL
  INTEGER*4 RETCODE, SYSRET
  INTEGER*2 FILEID
END SUB
```

```
SUB MEMUNSYN (RETCODE,SYSRET,FILEID) EXTERNAL
  INTEGER*4 RETCODE, SYSRET
  INTEGER*2 FILEID
END SUB
```

```
SUB FREEMEM (RETCODE,SYSRET,FILEID) EXTERNAL
  INTEGER*4 RETCODE, SYSRET
  INTEGER*2 FILEID
END SUB
```



```

SUB AVAILMEM (KOUNT,SYSRET) EXTERNAL
    INTEGER*4 KOUNT, SYSRET
END SUB

SUB MEMWRITE (RETCODE,SYSRET,FILEID,INDATA$, \
              RECNUM,OFFSET,LENGTH) EXTERNAL
    INTEGER*4 RETCODE, SYSRET, RECNUM
    INTEGER*2 FILEID, OFFSET, LENGTH
    STRING    INDATA$
END SUB

SUB MEMREAD (RETCODE,SYSRET,FILEID,INDATA$, \
             RECNUM,OFFSET,LENGTH) EXTERNAL
    INTEGER*4 RETCODE, SYSRET, RECNUM
    INTEGER*2 FILEID, OFFSET, LENGTH
    STRING    INDATA$
END SUB

SUB MEMSRCHB (RETCODE,SYSRET,FILEID,INDATA$, \
             SLENGTH,TOFFSET,RLENGTH,ROFFSET) EXTERNAL
    INTEGER*4 RETCODE, SYSRET
    INTEGER*2 FILEID, SLENGTH, TOFFSET, RLENGTH, ROFFSET
    STRING    INDATA$
END SUB

SUB MEMSRCHS (RETCODE,SYSRET,FILEID,INDATA$, \
             SLENGTH,TOFFSET,RLENGTH,ROFFSET) EXTERNAL
    INTEGER*4 RETCODE, SYSRET
    INTEGER*2 FILEID, SLENGTH, TOFFSET, RLENGTH, ROFFSET
    STRING    INDATA$
END SUB

SUB MEMWRKEY (RETCODE, SYSRET, FILEID, INDATA$, \
             KLENGTH, LENGTH) EXTERNAL
    INTEGER*4 RETCODE, SYSRET
    INTEGER*2 FILEID, KLENGTH, LENGTH
    STRING    INDATA$
END SUB

SUB MEMDLKEY (RETCODE, SYSRET, FILEID, KEY$, \
             KLENGTH) EXTERNAL
    INTEGER*4 RETCODE, SYSRET
    INTEGER*2 FILEID, KLENGTH
    STRING    KEY$
END SUB

SUB MEMCLEAR (RETCODE, SYSRET, FILEID) EXTERNAL
    INTEGER*4 RETCODE, SYSRET
    INTEGER*2 FILEID
END SUB

```

*Table 41. Function return codes*

Return code	Description
-1009	Attempted to allocate more than 64 files
-1011	Out of memory; cannot continue
-1013	Attempted to allocate the same file ID twice
-1014	File not found, or attempted to use MEMSYN or MEMUNSYN on a non-shared file
-1015	Record position outside of the file
-1016	Search record not found
-1017	Data length not valid (greater than 512 bytes), negative length, or empty string passed as parameter to subroutine

Table 41. Function return codes (continued)

Return code	Description
-1019	Receiving string not large enough
-1020	Requested shared memory greater than 65,524 bytes in length
-1021	Null string parameter detected

---

## Terminal hard disk files

Terminal hard disk support is available for all supported 4694 models. The operating system automatically installs the hard disk driver and enables you to format the hard disk if the terminal hard disk is installed. See the *4690 OS: User's Guide* for instructions on formatting the terminal hard disk. The terminal's hard disk is drive C: and can be accessed by terminal applications. The operating system supports one hard disk in the terminal.

### Considerations for installing a hard disk in the terminal

Hard disk installation automatically occurs during the terminal IPL process if the hard disk is installed. Some characteristics of the hard disk are fixed and cannot be redefined. These characteristics include the following subdirectories created on hard disk:

- ADX\_SPGM
- ADX\_SDT1
- ADX\_IPGM
- ADX\_IDT1
- ADX\_UPGM
- ADX\_UDT1

### Accessing hard disk files from terminal applications

Your terminal application can call ADXCOPYF to copy files from the controller hard disk to the terminal hard disk.

#### ADXCOPYF (copying hard disk files)

You can use the subprogram originally designed for RAM disk files for the hard disk files in the terminal. See "ADXCOPYF (copying RAM disk files)" on page 315 for the function declaration, call, and return codes.

#### ADXDIR (listing terminal hard disk files)

Terminal applications use the ADXDIR subprogram to list the files on hard disk C: and to show the amount of free space on the disk. This subprogram displays the same type of information about terminal hard disk that the DIR command displays about the controller hard disks. The ADXDIR subprogram is in the system runtime subroutine library ADXUCRTL.L86 for medium memory model terminal applications and ADXUCLTL.L86 for big memory terminal applications. See "ADXDIR (listing terminal RAM disk files)" on page 322 for the function declaration, call, and return codes.

---

## Chapter 17. Designing Applications with Other Languages

This chapter contains coding guidelines and restrictions that you should consider when you are writing a program in a language other than 4680 BASIC to run under the operating system. It contains specific information about programming applications in C language and COBOL.

The interfaces described in this chapter provide access to many parts of the operating system. This chapter explains how to use these interfaces.

**Note:** A CBASIC or 16-bit C application cannot access long file name support either on the controller or from the terminal.

---

### Operating System Interfaces for C and COBOL

The operating system interfaces described in this chapter provide access to various operating system features for applications written in C language or COBOL. 4690 OS V2 or higher allows you to develop 32-bit applications using the C language. Because the operating system supports both 16-bit and 32-bit applications written in C language, differences between 16-bit coding and 32-bit coding are noted throughout the chapter. The layout and sizes of data structures passed into these functions are unchanged. Any existing code declaring these structures using "int" type variables must be changed because the size of this data type has increased from 16 to 32 bits. In addition, these structures need to be packed on one-byte boundaries to ensure that the data layout remains unchanged.

The functions described in this chapter can be used only with store controller applications. See Chapter 19, "Designing Terminal Applications With C Language," on page 397 for information on functions to be used with terminal applications.

#### 16-Bit

These functions are written in C language and can be used by applications written in C language and COBOL. The functions were compiled using a memory model that generates 32-bit addresses. The memory model permits multiple code and data segments. To use these functions, link with ADXAPACL.L86, which is located on the 4690 Optionals. Use the search option to link only the required code with the application and not the entire library. These functions were written in MetaWare High C. Any application that is not written in High C also must link with ADXAPABL.L86.

#### 32-Bit

The 32-bit versions of these functions are written in C language. They were compiled in a 32-bit "flat mode" programming model. In this model, all address pointers are 32-bits long. In addition, there is no need to worry about the sizes and number of code and data segments. To use these functions, include the file CAPIC.H and link with the CAPIC.LIB. Both of these are on the 4690 Optionals as part of the VADEVENV.ZIP file. See Chapter 22, "Creating 32-Bit Programs Using VisualAge C/C++," on page 665 for more information on creating 32-bit C/C++ programs.

In the 16-bit interface, parameters are assumed to be passed with the first parameter being pushed last on the stack. In the 32-bit interface, parameters are assumed to be passed via the Optlink calling convention used by the VisualAge C/C++ Compiler. The first parameter for most functions is a four-byte return code. A negative return code indicates an error code. Any nonsystem error codes that might be returned by these functions are listed with the function descriptions. System error codes are returned as four-byte hexadecimal values. See the *4690 OS: Messages Guide* for a complete list of system error codes. The conversion function can be used to convert the four-byte hexadecimal bytes to eight printable ASCII characters.

The bit numbering scheme numbers the bits right to left with the least significant bit (lsb) being bit zero and the most significant bit (msb), being bit 15. The following is an example of numbering for a two-byte word.

		BIT NUMBERING															
		msb														lsb	
Bits		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

## Using the 16-Bit C Language Interface

Applications written in C language can be written in Metaware High C or an equivalent C language. The functions in this chapter were tested with Metaware High C code. The test code was compiled using the big memory model. The big memory model allows multiple code and data segments of up to 64K bytes each and creates 32-bit addresses. An application using these functions must be compiled in the same manner.

The assumed sizes of the basic data types used are:

**char** = 1 byte  
**int** = 2 bytes  
**long** = 4 bytes

## Using the 32-Bit C Language Interface

Applications using the 32-bit C language can be written using the VisualAge C/C++ Compiler for Windows and be called from either C or C++. All pointers in 32-bit mode are 32-bits long. See Chapter 22, "Creating 32-Bit Programs Using VisualAge C/C++," on page 665 for more information on creating 32-bit C/C++ programs.

The assumed sizes of the basic data types in the 32-bit interface are:

**char** = 1 byte  
**short** = 2 bytes  
**int** = 4 bytes  
**long** = 4 bytes

## Using the COBOL Language Interfaces

Applications written in COBOL can be written in COBOL/2 or an equivalent COBOL language. The functions described here were tested with COBOL/2 code. The test code was compiled using the huge memory model and the following compiler directives:

- /NOIBMCOMP
- /LITLINK
- /VSC2
- /OS(FLEXOS)

An application using these functions must be compiled in the same manner. The HUGE memory model produces 32-bit addresses and *far calls*. It allows literal names if the compiler directive /LITLINK is used. The directive /NOIBMCOMP causes the compiler to store data in one byte binary fields. The directive /LITLINK causes the compiler to create external references to the literal names of the functions in this chapter. The directive /VSC2 allows the use of the BY VALUE in the CALL statement. If the application is written in COBOL/2, the directive /OS(FLEXOS) is needed for compatibility with the LINK86 command. If the directive OS(FLEXOS) is not recognized, a later level of COBOL/2 is required. COBOL/2 references FAR\_DATA in the OBJ files and OS(FLEXOS) removes it. If CBLINK.BAT is used to link files, the message "FAR\_DATA NOT FOUND" are generated and the resulting 286 file does not execute properly. To avoid this problem, edit the file MF\_LNK.INP that is used by CBLINK.BAT and remove ";class[FAR\_DATA,DATA]" from the data statement. Or you can use LINK86 with an INP file that does not contain FAR\_DATA.

For applications using these functions, the usage type, COMP-5, is assumed for numeric variables. This usage indicates that the value that can be stored for a numeric variable is not limited to the number of decimal digits specified in the picture clause. The value can be the largest binary number that can be stored in the indicated space.

The convention for byte length of COBOL numeric variables is:

**PIC 9(2)**        = 1 byte  
**PIC 9(4)**        = 2 bytes  
**PIC 9(8)**        = 4 bytes

In this chapter, some of the descriptions about how to call each function contain a variable that is listed as USAGE IS POINTER. The address of this pointer is coded BY VALUE in the CALL statement. These variables are specified in this manner to allow variation in applications and programming styles. The following is an example of how the variables can be defined in a program:

```
01 KFBUFF.  
   03 KF-KEY   PIC 9(8)   USAGE IS COMP-5.  
   03 KF-DAT   PIC X(46)  USAGE IS DISPLAY.  
77 BUFFADR    USAGE IS POINTER.  
PROCEDURE DIVISION.  
SET BUFFADR TO ADDRESS OF KFBUFF.  
CALL ... ... USING BY VALUE BUFFADR.
```

## Disk File Management

Disk media on the operating system contain the following:

- |                         |  |
|-------------------------|--|
| <b>File names</b>       | Every file created on a disk must have a name to identify the file. Unless a drive allows long file names, the operating system forces all file names to uppercase on the media.   |
| <b>File record size</b> | When a file is created, the record size must be specified. A record size of zero is equivalent to a record size of one byte. If a record size of zero or one is specified, the operating system does not perform record boundary checks when a READ or WRITE is requested. |

## File Access

Access to files is initiated using a CREATE or an OPEN operation. Use a CREATE to open a new file. Use an OPEN to gain access to an existing file. For both operations, a file name is specified and a four-byte file number is returned. The file number is then used for all subsequent operations on that file. When the file is closed the file number is deleted. If the file is reopened, a new number is assigned.

The DELETE statement removes a file from the disk directory. Files to be deleted are indicated by the file name. A file cannot be deleted when it is read-only or if it is currently open. A file can be automatically deleted by setting one of two flag bits when the file is created. One flag bit determines whether a file is to be treated as temporary or permanent. If a file is marked as temporary, it is deleted after the last open is closed. If a file is marked as a permanent file, the file remains after the last CLOSE. The other flag bit that can be selected when the file is created determines what action is to be taken if a file with the same name exists. If this flag bit is set, the existing file is deleted before the new file is created. If this bit is not set, an error is returned.

## Access Modes

Access modes determine if a file can be shared. If the file is shared, the following modes are selected when the file is opened:

- Exclusive access by calling process
- Allow reads by other processes
- Allow reads and writes by other processes

Exclusive access to a file in one process prevents any other process from sharing the file. Exclusive access to a file is denied if another process has the file open. If a process tries to open a file with WRITE privilege and the file has been opened in ALLOW-READS mode, then the OPEN is denied and an error is returned.

ALLOW/READ/WRITE mode has two options: shared or unique file pointer. The shared file pointer mode is only available to processes with the same family ID, and all processes in the family must specify this mode. For processes outside the family, the file appears in exclusive mode. There are no restrictions when the unique file pointer option is selected.

## File Pointers

The operating system supports both sequential and random access to disk files by using the file pointer. Sequential file access is supported by a file pointer that is incremented with each read or write to maintain the position within the file. Random file access is supported by an offset specified with each read or write call. The offset can be specified by the file pointer, the beginning of the file, or the end of the file.

The file pointer is initialized to zero when a file is created or opened. Subsequent READs and WRITEs move the file pointer to the byte position of the next sequential location. For example, if a new file is created and 12 bytes are written, the file pointer would be pointing at the 13th byte (essentially the EOF marker).

Separate processes sharing access to the same file can share the same file pointer or can have separate ones. File pointer sharing is limited to processes with the same family identification (FID) number. When the pointer is shared, reads or writes by any process update the file pointer. The seek function can be used to determine the file pointer's location or to position the file pointer.

**Note:** In multi-threaded 32-bit programs, file handles and pointers are shared between all threads in a process.

## Pipe Management

Two or more processes can communicate by using a type of file known as a pipe, which is supported through a special device known as *pi*. Creating a pipe establishes a buffer used for the deposit and withdrawal of messages. Pipe files have two ends, one to write into and the other to read from. Messages are deposited and withdrawn from the pipe on a first-in-first-out (FIFO) basis. The pipe length is the only limit to the number of messages you can store in a pipe at one time.

In all calls that require a pipe name, the pipe name must be preceded with the device name (*pi*) or a logical name can be defined that includes the *pi* reference.

## Creating and Deleting Pipes

Use the CREATE function to make a pipe. All pipes are handled in the same manner as sequential files. The CREATE parameters are used as follows:

- Set the flags to request READ, WRITE, or DELETE privileges and to request the access mode. The privileges have the same meaning for pipes as they do for disk files.
- The pipe name must include the device name, (*pi*) plus up to eight (8) alphanumeric characters. Pipe names are case sensitive. The default is lowercase.
- The record size parameter controls the message blocks. For example, if a record size of four is specified, all pipe I/O is conducted in four-byte blocks. Record size of zero or one must be specified if the application uses the WAIT function with the pipe.
- Set the size to the pipe buffer length. The size is independent of the message length but must be a multiple of the record size.

Use a DELETE to remove a pipe. A pipe that is marked as temporary when it is created is automatically deleted on the last CLOSE. If a pipe marked as temporary is used to communicate between two processes, the pipe is deleted automatically from the system when the processes terminate because files are automatically closed when a process ends.

## Pipe Access

Pipe access privileges are affected by existing access modes. The following rules govern the privileges available:

- A process' open access is never restricted by an open connection previously made by the same process.
- The READ and WRITE ends of a pipe are considered separate with respect to open restrictions. For example, an OPEN with exclusive READ does not restrict a process from opening a pipe as shared WRITE.
- Any exclusive OPEN prevents other access requests to the same end.
- A shared OPEN prevents other exclusive access requests but allows other shared requests to the same end.
- A shared file pointer request restricts pipe access to processes with the same FID. All processes sharing the pipe must select the shared file pointer mode. A process that requests a different mode is denied access. For processes outside the family, the request functions as an exclusive request.

A pipe is used differently if an end is opened in exclusive or shared mode. If one end of a pipe is opened in exclusive mode and then closed, a READ or WRITE attempt to the other end results in an EOF error. It does not matter how the other end was opened.

If one end of a pipe is opened in shared mode and then closed, the operating system uses the pipe as if it were still open on the other end. Therefore, any process accessing the pipe waits until the operation is complete. A pipe opened in shared file pointer mode is shared only by those processes with the same FID. Notice the distinction between shared mode and shared file pointer mode. If one end of a pipe is opened in shared file pointer mode, and then the pipe is closed by all of the processes accessing that end, any processes accessing the other end receive an EOF error.

**Interprocess Communications:** The READ and WRITE functions operate the same way for pipes as for files and the READ and WRITE flags and parameters are used in the same way. Any number of processes can participate in the exchange.

The amount of data written to and read from the pipe is independent of the pipe size. The following procedures are observed when the amount exceeds the size:

- On WRITES when the pipe is full, the process waits for another process to read data from the other end. When the reading process removes enough to allow the data to be written, the operation completes.
- On READs when the pipe is empty, the process waits for another process to write data to the other end. The operation completes when enough data has been written to satisfy the READ request.

A READ request issued when there is no data in the pipe causes the process to wait until there is enough data available in the pipe to satisfy the read request. To avoid a possible hang, use the WAIT operation to wait for a specified time. Then the application can select whether or not to wait again if the time limit expires before data is available to read.

**Synchronization and Exclusion:** A pipe can be created with a zero-length buffer size for use as a simple semaphore. For semaphore pipes, a READ operation obtains the pipe and a WRITE releases it. If another process has obtained the pipe previously, the calling process waits until a WRITE to that pipe has been performed. WRITE operations, on the other hand, never wait; if the pipe was released previously, the call returns without an error.

**Nondestructive Read:** The information stored in a pipe can be previewed using the READ operation by setting the specific flag bit to request a nondestructive read. This allows a pipe to be used as a common data area among multiple applications. It also allows an application to pre-read a length field or message type field within a message and then read the complete message at a later time. To read records of varying lengths, specify record length of one when the pipe is created so that the operating system does not perform record checking; otherwise, an invalid record size error is returned.



**Attention:** Nondestructive reads can be dangerous if there are multiple readers of a pipe. It is the responsibility of the application to handle synchronization of pipe usage when there are multiple processes involved.

**Pipe Routing Services:** Pipe Routing Services (PRS) is used for communications between a store controller and a terminal. The WAIT operation available is the same as for regular pipes. PRS enables applications to exchange data with applications in other store controllers or terminals by using pipes. These pipes are identified by a pipe ID, which is a letter between A and Z. Each store controller or terminal can have up to 26 IDs (A through Z). Each ID in a store controller or a terminal must be unique. For example, if several applications are running in a store controller at once, each must use a different pipe ID.

On the store controller, pipes are treated like sequential files. A PRS pipe must first be created. Then the application should wait for data to be available before requesting a READ. For PRS pipes in the store controller, the READ buffer can be large but in the terminal the limit is 240 bytes. The maximum size for all PRS messages is 120 bytes.

To write to a PRS pipe, the PRS driver must be initialized. This initialization must be requested only once for each load of an application. After the driver is initialized, a write can be performed. All PRS pipes are temporary. A pipe is deleted when the last process that has access to it has ended. The PRS functions described for C language and COBOL can only be used in store controller applications.

## File and Pipe Services

This section contains examples of C and COBOL interfaces for keyed, non-keyed, and direct files. It also contains examples for pipes.

### Create Point-of-Sale Keyed File

Creating a point-of-sale keyed file sets up the file as a keyed file and write the keyed file control block as the first record. See “Keyed File Control Record” on page 194 for a description of the keyed file control block. The file is opened if no error occurred. The file must be closed if no access is needed.

**Note:** If you create a local file and the Delete file if already exists option is selected and a distributed file with the same name already exists, only the prime copy is deleted. The image copy remains.

#### 16-Bit C Interface:

```
void ADX_CC_CREATE_KFILE(long *fnum, unsigned int flags, char *filename,
long filesize, unsigned int *buffadr, long buffsize);
```

#### 32-Bit C Interface:

```
void ADX_CC_CREATE_KFILE(long *fnum, unsigned short flags, char *filename,
long filesize, unsigned short *buffadr, long buffsize);
```

#### COBOL Interface:

```
77 FNUM          PIC S9(8)      USAGE IS COMP-5.
77 FLAGS         PIC 9(4)       USAGE IS COMP-5.
77 FILENAME      PIC X(n)       USAGE IS DISPLAY.
*      n = length of FILE-NAME including
      terminating NULL or blank.
77 FILESIZE      PIC 9(8)       USAGE IS COMP-5.
77 BUFFADR       PIC 9(8)       USAGE IS POINTER.
77 BUFFSIZE      PIC 9(8)       USAGE IS COMP-5.
```

```
CALL "ADX_CC_CREATE_KFILE" USING FNUM,
BY VALUE FLAGS, BY REFERENCE FILENAME,
BY VALUE FILESIZE, BY VALUE BUFFADR, BY VALUE BUFFSIZE.
```

#### Parameters:



**fnum** Return code. It contains the file number if no error occurred. The file is automatically opened. The calling process must close the file if no access is needed. If an error occurred, the error code is returned and a file is **not** created.

**flags**

- bit 0:**
- 1 = Delete file
  - 0 = No delete
- bit 1:** Reserved (must be 0)
- bit 2:**
- 1 = Write
  - 0 = No write
- bit 3:**
- 1 = Read
  - 0 = No read
- bit 4:**
- 1 = Shared
  - 0 = Exclusive
- bit 5:**
- 1 = Allow shared commands if shared
  - 0 = Allow shared READ or WRITE commands if shared
- bit 6–7:** Reserved (must be zero)
- bit 8:**
- 1 = Temporary—delete on last close
  - 0 = Permanent
- bit 9:** Reserved (must be zero)
- bit 10:**
- 1 = Delete file if it already exists
  - 0 = Return error if file exists
- bit 11 to 15:** Reserved (must be zero)

**filename**

Address of a NULL or blank terminated name string, or a previously defined logical name. If the file is not in the current directory, the string must include the path specification. The maximum length is 128 bytes including the NULL or blank.

**Note:** When using the 32-bit API to access files on a drive enabled for long filename support, the file name is considered to end at the first blank. The maximum length of a long file name is 260 bytes including the NULL or blank.

**filesize**

The size must be derived from the following algorithm and rounded up to the nearest 512-byte multiple. The maximum size for a keyed file record is 508.

**T** = Total number of records

**L** = Logical keyed record size

**NL** =  $508/L$  (integer division)

If (T divided by NL has a remainder)

**size** =  $512 + (512 \times (1 + T/NL))$

Else (no remainder)

**size** =  $512 + (512 \times (T/NL))$

The total number of records should be at least 20% greater than the maximum number of records expected to account for the way the records must be distributed in the file and to allow for future growth.

**buffadr**

The address of the buffer containing keyed file information (BUF14 or BUF18 (see bit 11 of pflags)).

**buffsize**

The size of buffer (BUF14 size = 14. BUF18 size = 18.)

BUF14 — Information buffer for creating a keyed file less than 32 MB. Size = 14 bytes. Figure 16 on page 340 illustrates how the bytes are allocated if the *buffsize* is 14.

BUF14	
Byte	
0	pflags
2	numblocks
4	randivsr
6	keyrecl
8	keylen
10	cthresh
12	Reserved

Figure 16. 14-Byte Buffsize

BUF18 — Information buffer for creating a keyed file greater than or equal to 32 MB. Size = 18 bytes. Figure 17 on page 341 illustrates how the bytes are allocated if the *buffsize* is 18.

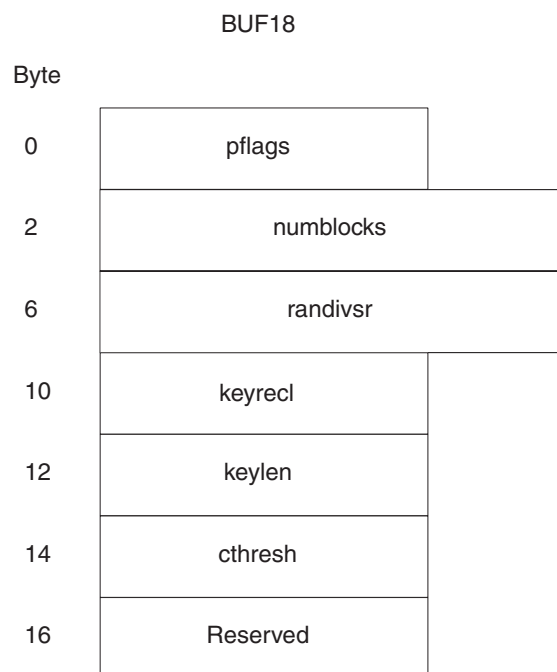


Figure 17. 18-Byte Buffsize

### pflags

**bit 0:** 1 = Keyed file create

**bit 1 to 3:**  
0 = Reserved (must be zero)

**bit 4:**  
1 = File is mirrored (see note 1)  
0 = File is not mirrored

**bit 5:**  
1 = File is compound (see note 1)  
0 = File is not compound

**bit 6:** 0 = Reserved (must be zero)

**bit 7:**  
1 = File is local only (see note 1)  
0 = File is not local

**bit 8:**  
1 = Distributed at close (see note 2)  
0 = No Distribution at close

**bit 9:**  
1 = Distribution per update (see note 2)  
0 = No Distribution per update

**bit 10:** Reserved (must be zero)

**bit 11:**  
1 = This information structure is for files greater than or equal to 32 MB. (See format BUF18.)  
0 = This information structure is for files less than 32 MB. (See format BUF14.)

**bit 12:**

1 = The data blocks are not cleared to NULLs. (If this option is used, the user application must clear all data blocks to NULLs before adding records to the file.)  
 0 = All data blocks are cleared to NULLs

**bits 13 to 15:**

0 Reserved

**Notes:**

1. Bits 4, 5, and 7 are mutually exclusive (only one of the three types can be chosen).
2. Bits 8 and 9 are mutually exclusive (only one of the two types can be chosen).

**numblocks**

Number of 512-byte physical records in the file. Must be = filesize (as calculated previously) / 512.

**randivsr**

File randomizing divisor.

This must be greater than zero and less than numblocks. This value determines how the keyed records are dispersed in the file.

**keyrecl**

Logical keyed record size. Must be in range 1 to 508.

**keylen**

Length of record key found in logical keyed records.

**cthresh**

Chaining threshold value. Must be in range 1 to 255. The value is the number of sectors that are checked to find a keyed record before a warning is sent that a file is becoming inefficient. The application is not notified that the limit has been exceeded, and data is still returned to the application.

See the *4690 OS: Messages Guide* for system errors.

**Open Keyed File**

This operation opens a keyed file.

**16-Bit C Interface:**

```
void ADX_COPEN_KFILE(long *fnum, unsigned int flags, char *filename,
unsigned int *parmbuff, long pbufsize);
```

**32-Bit C Interface:**

```
void ADX_COPEN_KFILE(long *fnum, unsigned short flags, char *filename,
unsigned short *parmbuff, long pbufsize);
```

**COBOL Interface:**

```
77 FNUM          PIC S9(8)          USAGE IS COMP-5.
77 FLAGS         PIC 9(4)           USAGE IS COMP-5.
77 FILENAME      PIC X(n)           USAGE IS DISPLAY.
*  n=length of FILE-NAME including terminating NULL or blank.
77 PARMBUFF      PIC 9(8)           USAGE IS POINTER.
77 PBUFSIZE      PIC 9(8)           USAGE IS COMP-5.
```

```
CALL "ADX_COPEN_KFILE" USING FNUM, BY VALUE FLAGS,
BY REFERENCE FILENAME, BY VALUE PARMBUFF, BY VALUE PBUFSIZE.
```

## Parameters:

### flags

<b>bit 0:</b>	1 = Delete file 0 = No delete
<b>bit 1:</b>	1 = Execute 0 = No execute
<b>bit 2:</b>	1 = Write 0 = No write
<b>bit 3:</b>	1 = Read 0 = No read
<b>bit 4:</b>	1 = Shared 0 = Exclusive
<b>bit 5:</b>	1 = Allow shared reads if shared 0 = Allow shared read or write if shared
<b>bits 6 to 15:</b>	Reserved (must be zero)

### filename

Address of NULL or blank terminated name string or a previously defined logical name. If the file is not in the current directory, the string must include the path specification. The maximum length is 128 bytes, including the NULL or blank.

**Note:** When using the 32-bit API to access files on a drive enabled with long file name support, the file name is still considered to end at the first blank. The maximum length of a long file name is 260 bytes including the NULL or blank.

### parmbuff

User address where the keyed record size and key length are returned. The record size is in the first two bytes of the buffer and the key length is in the second two bytes of the buffer. The buffer must be at least four bytes in size. If pbufsize = 0, no values are returned.

### pbufsize

Size of parmbuff, in bytes. If pbufsize = 0, no values are returned in parmbuff.

**fnum** Return code. It contains the file number if no error occurred. If an error occurred, the file is closed and an error code is returned.

See the *4690 OS: Messages Guide* for system errors.

## Close Keyed File

This operation closes a keyed file and frees all associated buffer space. A partial CLOSE flushes the associated I/O buffer but leaves the file open.

### 16-Bit C Interface:

```
void ADX_CCLOSE_KFILE(long *ret, unsigned int option, unsigned int flags,  
long fnum)
```

### 32-Bit C Interface:

```
void ADX_CCLOSE_KFILE(long *ret, unsigned short option, unsigned short  
flags, long fnum)
```

### **COBOL Interface:**

77	RET	PIC S9(8)	USAGE IS COMP-5.
77	OPTION	PIC 9(4)	USAGE IS COMP-5.
77	FLAGS	PIC 9(4)	USAGE IS COMP-5.
77	FNUM	PIC S9(8)	USAGE IS COMP-5.

CALL "ADX\_CCLOSE\_KFILE" USING RET,  
BY VALUE OPTION, BY VALUE FLAGS, BY VALUE FNUM.

### **Parameters:**

#### **option**

0 = close file  
1 = zero and close file (only valid for keyed files)

#### **flags**

0 = full close  
1 = partial close (flush only)

**fnum** File number of file to be closed, as returned from an OPEN or CREATE.

**ret** Return code. An error code is returned if an error occurred.

See the *4690 OS: Messages Guide* for system errors.

## **Read Keyed Record**

This operation reads data from the indicated record in the specified keyed file.

### **16-Bit C Interface:**

```
void ADX_CREAD_KREC(long *nbytes, unsigned int option, long fnum, char *buffadr,  
long buffsize, long keylen);
```

### **32-Bit C Interface:**

```
void ADX_CREAD_KREC(long *nbytes, unsigned short option, long fnum, char *buffadr,  
long buffsize, long keylen);
```

### **COBOL Interface:**

77	BYTES	PIC S9(8)	USAGE IS COMP-5.
77	OPTION	PIC 9(4)	USAGE IS COMP-5.
77	FNUM	PIC S9(8)	USAGE IS COMP-5.
77	BUFFADR		USAGE IS POINTER.
77	BUFFSIZE	PIC 9(8)	USAGE IS COMP-5.
77	KEYLEN	PIC 9(8)	USAGE IS COMP-5.

CALL "ADX\_CREAD\_KREC" USING BYTES,  
BY VALUE OPTION, BY VALUE FNUM, BY VALUE BUFFADR,  
BY VALUE BUFFSIZE, BY VALUE KEYLEN.

### **Parameters:**

#### **option**

0 = Read no lock  
1 = Read and lock

The record is locked before it is read. The application program must wait for the record to become available.

**fnum** File number of file from which to read data. The file must be activated by a CREATE or OPEN before requesting a READ.

**buffadr** Address of buffer in which to put data that is read. The key for the record to read must begin at offset zero in this buffer.

<b>buffsize</b>	Number of bytes to read. The size must equal the record length specified when the file was created.
<b>keylen</b>	Length of the key string passed in the read buffer. This value must equal the key length specified when the keyed file was created.
<b>nbytes</b>	Return code. It contains the number of bytes read if no error occurred. An error code is returned if an error occurred.

See the *4690 OS: Messages Guide* for system errors.

## Write Keyed Record

This operation writes a keyed record in the specified file.

### 16-Bit C Interface:

```
void ADX_CWRITE_KREC(long *nbytes, unsigned int option, unsigned int flags,
long fnum, char *bufaddr, long buffsize);
```

### 32-Bit C Interface:

```
void ADX_CWRITE_KREC(long *nbytes, unsigned short option, unsigned short flags,
long fnum, char *bufaddr, long buffsize);
```

### COBOL Interface:

77	NBYTES	PIC S9(8)	USAGE IS COMP-5.
77	OPTION	PIC 9(4)	USAGE IS COMP-5.
77	FLAGS	PIC 9(4)	USAGE IS COMP-5.
77	FNUM	PIC S9(8)	USAGE IS COMP-5.
77	BUFADDR		USAGE IS POINTER.
77	BUFFSIZE	PIC 9(8)	USAGE IS COMP-5.

```
CALL "ADX_CWRITE_KREC" USING NBYTES,
BY VALUE OPTION, BY VALUE FLAGS, BY VALUE FNUM,
BY VALUE BUFADDR, BY VALUE BUFFSIZE.
```

### Parameters:

#### option

##### bit 0:

0 = Unlock = no  
1 = Unlock = yes

The record is unlocked after being written.

##### bit 1:

0 = Do not hold the write  
1 = Hold the write

The hold flag prevents the data from being physically written to a disk file until the next write with hold option is issued by this process.

**bits 2 to 15:** Must be zero

**flags** Must be zero

**fnum** File number where data is to be written, as returned from a CREATE or an OPEN.

**bufaddr** Address of buffer containing data to write. The buffer must contain the appropriate record's key at offset 0.

**buffsize** Number of bytes to write.

**nbytes** Return code. It contains the number of bytes written if no error occurred. Notice that a

zero (0) return value is a special case for the WRITE with HOLD and is considered a correct value. An error code is returned if an error occurred.

See the *4690 OS: Messages Guide* for system errors.

## Delete Keyed Record

This operation deletes a specified record from a keyed file.

### 16-Bit C Interface:

```
void ADX_CDELETE_KREC(long *ret, long fnum, char *buffadr, long buffsize);
```

### 32-Bit C Interface:

```
void ADX_CDELETE_KREC(long *ret, long fnum, char *buffadr, long buffsize);
```

### COBOL Interface:

77	FNUM	PIC S9(8)	USAGE IS COMP-5.
77	BUFFADR		USAGE IS POINTER.
77	BUFFSIZE	PIC 9(8)	USAGE IS COMP-5.
77	RET	PIC S9(8)	USAGE IS COMP-5.

```
CALL "ADX_CDELETE_KREC" USING RET,  
    BY VALUE FNUM, BY VALUE BUFFADR, BY VALUE BUFFSIZE.
```

### Parameters:

<b>fnum</b>	File number of file containing record to delete, as returned by an OPEN or CREATE.
<b>buffadr</b>	Address of buffer containing the key of the record to be deleted.
<b>buffsize</b>	Length of key string pointed to by <i>buffadr</i> .
<b>ret</b>	Return code. It contains an error code if an error occurred.

See the *4690 OS: Messages Guide* for system errors.

## Create Point-of-Sale Non-Keyed File

This operation creates a point-of-sale nonkeyed file. A point-of-sale file is different from any other file because it can be distributed to other controllers as a mirrored or a compound file. After the file is created, it is opened if no error occurred. The file must be closed if no access is needed.

**Note:** If you create a local file and the Delete file if already exists option is selected and a distributed file with the same name already exists, only the prime copy is deleted. The image copy remains.

### 16-Bit C Interface:

```
void ADX_CC_CREATE_POSFILE(long *fnum, unsigned int flags, char  
*filename, long filesize,  
unsigned int recsize, unsigned int ftype, unsigned int fdistrib);
```

### 32-Bit C Interface:

```
void ADX_CC_CREATE_POSFILE(long *fnum, unsigned short flags, char  
*filename, long filesize,  
unsigned short recsize, unsigned short ftype, unsigned short fdistrib);
```

### COBOL Interface:

77	FLAGS	PIC 9(4)	USAGE IS COMP-5.
77	FILENAME	PIC X(n)	USAGE IS DISPLAY.
*	n = length of FILE-NAME including terminating NULL or blank.		
77	FILESIZE	PIC 9(8)	USAGE IS COMP-5.
77	RECSIZE	PIC 9(4)	USAGE IS COMP-5.



```

77 FTYPE          PIC 9(4)          USAGE IS COMP-5.
77 FDISTRIB       PIC 9(4)          USAGE IS COMP-5.
77 FNUM           PIC S9(8)         USAGE IS COMP-5.

```

```

CALL "ADX_CC_CREATE_POSFILE" USING FNUM,
BY VALUE FLAGS, BY REFERENCE FILENAME, BY VALUE FILESIZE,
BY VALUE RECSIZE, BY VALUE FTYPE, BY VALUE FDISTRIB.

```

## Parameters:

### flags

#### bit 0:

1 = Delete file  
0 = No delete

#### bit 1:

Reserved (must be zero)

#### bit 2:

1 = Write  
0 = No write

#### bit 3:

1 = Read  
0 = No read

#### bit 4:

1 = Shared  
0 = Exclusive

#### bit 5:

- 1 = Allow shared READ commands if shared
- 0 = Allow shared READ or WRITE commands if shared

#### bit 6:

- 1 = Shared file pointer
- 0 = Unique file pointer

#### bit 7:

Reserved (must be zero)

#### bit 8:

- 1 = Temporary—delete on last close
- 0 = Permanent

#### bit 9:

Reserved (must be zero)

#### bit 10:

1 = Delete file if it already exists  
0 = Return error if file exists

**bits 11 to 15:** Reserved (must be zero)

### filename

Address of NULL or blank terminated name string or a previously defined logical name. If the file is not in the current directory, the string must include the path specification. The maximum length is 128 bytes, including the NULL or blank.

**Note:** When using the 32-bit API to access files on a drive enabled for long file name support, the file name is still considered to end at the first blank. The maximum length of a long file name is 260 bytes including the NULL or blank.

### filesize

File size in bytes

<b>recsize</b>	The READ, WRITE, and LOCK operations use this value to ensure that the requested action falls on record boundaries. Use a record size of zero or one if you do not want boundary checks performed.
<b>ftype</b>	File distribution type 0 = Local only 1 = Mirrored 2 = Compound file
<b>fdistrib</b>	File distribution method 0 = Distribute at close 1 = Distribution per update
<b>fnum</b>	Return code. It contains the file number if no error occurred. The file is automatically opened; therefore, the calling process must close the file if no access is needed. If an error occurs, the file is closed and the error code is returned.

See the *4690 OS: Messages Guide* for system errors.

## Create Non-Point-of-Sale File or Pipe

This operation creates a non-point-of-sale file or a pipe. The file is opened if no error occurred and it should be closed if no access is needed.

**Note:** If you create a local file and the Delete file if already exists option is selected and a distributed file with the same name already exists, only the prime copy is deleted. The image copy remains.

### 16-Bit C Interface:

```
void ADX_CC_CREATE_FILE(long *fnum, unsigned int flags, char *filename,
long filesize, unsigned int recsize);
```

### 32-Bit C Interface:

```
void ADX_CC_CREATE_FILE(long *fnum, unsigned short flags, char
*filename, long filesize, unsigned short recsize);
```

### COBOL Interface:

```
77  FLAGS          PIC 9(4)          USAGE IS COMP-5.
77  FILENAME       PIC X(n)          USAGE IS DISPLAY.
*   n = length of FILE-NAME including
*   terminating NULL or blank.
77  RECSIZE        PIC 9(4)          USAGE IS COMP-5.
77  FILESIZE       PIC 9(8)          USAGE IS COMP-5.
77  FNUM           PIC S9(8)         USAGE IS COMP-5.
```

```
CALL "ADX_CC_CREATE_FILE" USING FNUM,
BY VALUE FLAGS, BY REFERENCE FILENAME,
BY VALUE FILESIZE, BY VALUE RECSIZE.
```

### Parameters:

#### flags

##### bit 0:

1 = Delete file  
0 = No delete

**bit 1:** Reserved (must be zero)

##### bit 2:

1 = Write  
0 = No write

##### bit 3:

	1 = Read 0 = No read
<b>bit 4:</b>	1 = Shared 0 = Exclusive
<b>bit 5:</b>	1 = Allow shared READ commands if shared 0 = Allow shared READ or WRITE commands if shared
<b>bit 6:</b>	1 = Shared file pointer (disk files only) 0 = Unique file pointer
<b>bit 7:</b>	Reserved (must be zero)
<b>bit 8:</b>	1 = Temporary—delete on last close 0 = Permanent
<b>bit 9:</b>	Reserved (must be zero)
<b>bit 10:</b>	1 = Delete file if it already exists 0 = Return error if file exists
<b>bit 11 and 12:</b>	Reserved (must be zero)
<b>bit 13:</b>	1 = Force case to media default (pipes only) 0 = Do not force case on name
<b>bit 14 and 15:</b>	Reserved (must be zero)
<b>filename</b>	Address of NULL or blank terminated name string or a previously defined logical name. If the file is not in the current directory, the string must include the path specification. The maximum length is 128 bytes, including the NULL or blank. The name for a pipe must include <i>pi</i> : either in this string or in the logical name definition prefix.  <b>Note:</b> When using the 32-bit API to access files on a drive enabled for long file name support, the file name is still considered to end at the first blank. The maximum length of a long file name is 260 bytes including the NULL or blank.
<b>recsize</b>	The READ, WRITE, and LOCK operations use this value to make sure that the requested action falls on record boundaries. Use a record size of zero or one for files if no boundary checks are wanted, or for pipes if the WAIT function is used.
<b>filesize</b>	File size in bytes
<b>fnum</b>	Return code. It contains the file number if no error occurred. The file is automatically opened and the calling process must close the file if no access is needed. If an error occurred, the file is closed and the error code is returned.

See the *4690 OS: Messages Guide* for system errors.

## Open File or Pipe

This operation opens a point-of-sale nonkeyed file, non-point-of-sale file, or a pipe. Use Open Keyed File option to open a keyed file. The file pointer is initialized to zero when the file is opened.

### 16-Bit C Interface:

```
void ADX_COPEN_FILE(long *fnum, unsigned int flags, char *filename);
```

### 32-Bit C Interface:

```
void ADX_COPEN_FILE(long *fnum, unsigned short flags, char *filename);
```

### COBOL Interface:

```
77 FNUM          PIC S9(8)          USAGE IS COMP-5.  
77 FLAGS         PIC 9(4)           USAGE IS COMP-5.  
77 FILENAME      PIC X(n)           USAGE IS DISPLAY.  
*  n = length of FILE-NAME including terminating NULL or blank
```

```
CALL "ADX_COPEN_FILE" USING FNUM, BY VALUE FLAGS,  
BY REFERENCE FILENAME.
```

### Parameters:

#### flags

##### bit 0:

1 = Delete file  
0 = No delete

##### bit 1:

1 = Execute  
0 = No execute

##### bit 2:

1 = Write  
0 = No write

##### bit 3:

1 = Read  
0 = No read

##### bit 4:

1 = Shared  
0 = Exclusive

##### bit 5:

1 = Allow shared reads if shared  
0 = Allow shared R/W if shared

##### bit 6:

1 = Shared file pointer (disk files only)  
0 = Unique file pointer

**bits 7 to 12:** Reserved (must be zero)

##### bit 13:

1 = Force case to media default (pipes only)  
0 = Do not affect name case

**bit 14 and 15:** Reserved (must be zero)

#### filename

Address of NULL or blank terminated name string, or a previously defined logical name. If the file is not in the current directory, the string must include the path specification. The maximum length is 128 bytes, including the NULL or blank. The name for a pipe must include *pi*: either in this string or in the logical name definition prefix.

**Note:** When using the 32-bit API to access files on a drive enabled for long file name support, the file name is still considered to end at the first blank. The maximum length of a long file name is 260 bytes including the NULL or blank.

#### fnum

Return code. It contains the file number if no error occurred. If an error occurred, the file is closed and the error code is returned.

See the *4690 OS: Messages Guide* for system errors.

## Close File or Pipe

This operation closes a file or a pipe and frees all associated buffer space. A partial CLOSE flushes the associated I/O buffer but leaves the file open.

### 16-Bit C Interface:

```
void ADX_CCLOSE_FILE(long *ret, unsigned int flags, long fnum);
```

### 32-Bit C Interface:

```
void ADX_CCLOSE_FILE(long *ret, unsigned short flags, long fnum);
```

### COBOL Interface:

77	RET	PIC S9(8)	USAGE IS COMP-5.
77	FLAGS	PIC 9(4)	USAGE IS COMP-5.
77	FNUM	PIC S9(8)	USAGE IS COMP-5.

```
CALL "ADX_CCLOSE_FILE" USING RET, BY  
VALUE FLAGS, BY VALUE FNUM.
```

### Parameters:

#### flags

0 = Full close  
1 = Partial close

**fnum** File number of file to be closed, as returned from an OPEN or CREATE.

**ret** Return code. It contains an error code if an error occurred.

See the *4690 OS: Messages Guide* for system errors.

## Read Record from File or Pipe

This operation reads data from the indicated record in a specified non-keyed file or a pipe file. The file pointer is updated on every READ to be the byte position immediately following the last data byte that was read.

### 16-Bit C Interface:

```
void ADX_CREAD_REC(long *nbytes, unsigned int flags, long fnum, char *buffadr,  
long buffsize, long boffset);
```

### 32-Bit C Interface:

```
void ADX_CREAD_REC(long *nbytes, unsigned short flags, long fnum, char *buffadr,  
long buffsize, long boffset);
```

Example:

### COBOL Interface:

77	NBYTES	PIC S9(8)	USAGE IS COMP-5.
77	FLAGS	PIC 9(4)	USAGE IS COMP-5.
77	FNUM	PIC S9(8)	USAGE IS COMP-5.
77	BUFFADR		USAGE IS POINTER.
77	BUFFSIZE	PIC 9(8)	USAGE IS COMP-5.
77	BOFFSET	PIC S9(8)	USAGE IS COMP-5.

```
CALL "ADX_CREAD_REC" USING NBYTES,  
BY VALUE FLAGS, BY VALUE FNUM, BY VALUE BUFFADR,  
BY VALUE BUFFSIZE, BY VALUE BOFFSET.
```

### Parameters:

## flags

### bit 0:

1 = Read from device  
0 = Read from internal buffers

**bit 1:** Reserved (must be zero)

### bit 2:

1 = Nondestructive read  
0 = Normal read

**bits 3 to 7:** Reserved (must be zero)

**bits 8 and 9:** Interpretation of offset field

00 = Relative to beginning of file  
01 = Relative to file pointer (disk file only)  
10 = Relative to end of file (disk file only)

**bits 10 to 15** Reserved (must be zero)

**fnum** File number from which to read data, as returned from OPEN or CREATE.

**buffadr** Address of buffer in which to put data that is read.

**buffsize** Number of bytes to READ.

**boffset** Byte offset to begin reading, relative to the position indicated by flag bits 8 and 9. Negative offsets are allowed. Offset used for disk files only; set = 0 for pipes.

**nbytes** Return code. It contains the number of bytes read if no error occurred. Where *nbytes* is positive and not equal to *buffsize*, an end-of-file was encountered. An error code is returned if an error occurred. For pipes, if the buffer is empty, this operation waits for enough data to be written in the buffer to satisfy the read request.

See the *4690 OS: Messages Guide* for system errors.

## Write a Record to a File (nonkeyed) or Pipe

This operation writes data into the indicated record of a specified nonkeyed file or a pipe. See "Write Keyed Record" for keyed files. The file pointer is updated on every WRITE to be the byte position immediately following the last data byte that was written.

### 16-Bit C Interface:

```
void ADX_CWRITE_REC(long *nbytes, unsigned int option, unsigned int flags,  
long fnum, char *buffadr, long buffsize, long boffset);
```

### 32-Bit C Interface:

```
void ADX_CWRITE_REC(long *nbytes, unsigned short option, unsigned short flags,  
long fnum, char *buffadr, long buffsize, long boffset);
```

### COBOL Interface:

77	BYTES	PIC S9(8)	USAGE IS COMP-5.
77	OPTION	PIC 9(4)	USAGE IS COMP-5.
77	FLAGS	PIC 9(4)	USAGE IS COMP-5.
77	FNUM	PIC S9(8)	USAGE IS COMP-5.
77	BUFFADR		USAGE IS POINTER.
77	BUFFSIZE	PIC 9(8)	USAGE IS COMP-5.
77	BOFFSET	PIC S9(8)	USAGE IS COMP-5.

```
CALL "ADX_CWRITE_REC" USING BYTES,  
BY VALUE OPTION, BY VALUE FLAGS, BY VALUE FNUM,  
BY VALUE BUFFADR, BY VALUE BUFFSIZE, BY VALUE BOFFSET.
```

## Parameters:

### option

- 0 = Do not hold the write.
- 1 = Hold the write (not valid for pipes).

The Hold flag prevents the data from being physically written to a disk file until the next write with hold option is issued by this process. Each record must be less than or equal to 512 bytes in length when using the HOLD option.

### flags

#### bit 0:

- 1 = Flush buffers after write (disk file only).
- 0 = Do not flush buffers.

This forces the data to the media. If this is a zero length request, the media is updated with any pending writes.

**bits 1: 1 =** Truncate the file at the position specified in boffset. NOTE: The bufsize must be zero when this flag is specified.

**bits 2 to 7: =** Reserved (should be zero).

**bits 8 and 9:** Determine how the offset field is interpreted:  
00 = Relative to beginning of file  
01 = Relative to file pointer (disk file only)  
10 = Relative to end of file (disk file only)

**bits 10 to 15:** Reserved (must be zero)

**fnum** File number where data is to be written, as returned from an OPEN or CREATE.

**buffadr** Address of buffer containing data to write.

**bufsize** Number of bytes to write.

**boffset** Offset into file to start writing, depending on bits 8 and 9 of flags. Offset is used for disk files only; set = 0 for pipes.

**nbytes** Return code. It contains the number of bytes transferred if no error occurred. Note that a zero (0) return value is a special case for the WRITE with HOLD and is considered a correct value. Where *nbytes* is positive and not equal to *bufsize*, an end-of-media (disk or diskette full) condition exists. For pipes, if the buffer is full this operation waits until enough is read from the other end to allow the WRITE to complete. An error code is returned if an error occurred.

See the *4690 OS: Messages Guide* for system errors.

## Lock and Unlock Record

Access to records in a nonkeyed file are controlled by selectively locking and unlocking the record. LOCK the record before reading it, then UNLOCK the record after writing to it. The area to lock or unlock is determined from the offset and how it is used. This operation is only valid for disk files.

### 16-Bit C Interface:

```
void ADX_CLOCK_REC(long *ret, unsigned int flags, long fnum, long
boffset, long nbytes);
```

### 32-Bit C Interface:

```
void ADX_CLOCK_REC(long *ret, unsigned short flags, long fnum, long
boffset, long nbytes);
```

### **COBOL Interface:**

```
77  FLAGS          PIC 9(4)          USAGE IS COMP-5.
77  FNUM           PIC S9(8)         USAGE IS COMP-5.
77  BOFFSET        PIC S9(8)         USAGE IS COMP-5.
77  NBYTES         PIC S9(8)         USAGE IS COMP-5.
77  RET            PIC S9(8)         USAGE IS COMP-5.
```

```
CALL "ADX_CLOCK_REC" USING RET, BY VALUE FLAGS,
BY VALUE FNUM, BY VALUE BOFFSET, BY VALUE NBYTES.
```

### **Parameters:**

#### **flags**

##### **bits 0 and 1:** Lock Mode

00 = Unlock — release the indicated area

01 = Exclusive lock — prevents other processes from locking, reading from, or writing to the locked area

10 = Exclusive write lock — allows other processes to read the area but not to write to it or lock it

11 = Shared write lock — allows other processes to read the area and establish shared write lock but not to write to the area

##### **bits 2 and 3:** Reserved (must be zero)

##### **bit 4:**

1 = Return error on lock conflict

0 = Wait on lock conflict

##### **bits 5 to 7:** Reserved (must be zero)

##### **bits 8 and 9:** Interpretation of offset field

00 = Relative to beginning of file

01 = Relative to the pointer

10 = Relative to the end of file

##### **bits 10 to 15:** Reserved (must be zero)

**fnum** File number of file containing record to lock or unlock, as returned by a CREATE or OPEN.

**boffset** Offset of region to lock in file. See bits 8 and 9 in flags.

**nbytes** Length of region to lock, in bytes

**ret** Return code. An error code is returned if an error occurred.

See the *4690 OS: Messages Guide* for system errors.

### **Delete File or Pipe**

This operation deletes the designated file or pipe.

### **16-Bit C Interface:**

```
void ADX_DELETE_FILE(long *ret, unsigned int flags, char *filename);
```

### **32-Bit C Interface:**

```
void ADX_DELETE_FILE(long *ret, unsigned short flags, char *filename);
```

### **COBOL Interface:**

```
77  FLAGS          PIC 9(4)          USAGE IS COMP-5.
77  FILENAME        PIC X(n)         USAGE IS DISPLAY.
*   n = length of FILE-NAME including terminating NULL or blank.
```



```
77 RET PIC S9(8) USAGE IS COMP-5.
```

```
CALL "ADX_CDELETE_FILE" USING RET, BY VALUE FLAGS,  
BY REFERENCE FILENAME.
```

## Parameters:

### flags

1 = Force case to media default (pipes only).  
0 = Do not affect name case.

**filename** Address of NULL or blank terminated name string or a previously defined logical name. If the file is not in the current directory, the string must include the path specification. The maximum length is 128 bytes including the NULL or blank. The name for a pipe must include *pi*: either in this string or in the logical name definition prefix.

**Note:** When using the 32-bit API to access files on a drive enabled for long file name support, the file name is still considered to end at the first blank. The maximum length of a long file name is 260 bytes including the NULL or blank.

**ret** Return code. An error code is returned if an if error occurred.

See the *4690 OS: Messages Guide* for system errors.

## Rename a File

The rename operation changes the name of an existing disk file. If the file is currently open by another process, the file is not renamed and an error code is returned. If the new file name specifies another directory, the file is moved to that location. This feature is limited to directories on the same drive. Attributes, ownership, protection and date stamps are not changed.

### 16-Bit C Interface:

```
void ADX_CRENAME_FILE(long *ret, char *filename, char *newname);
```

### 32-Bit C Interface:

```
void ADX_CRENAME_FILE(long *ret, char *filename, char *newname);
```

### COBOL Interface:

```
77 FILENAME PIC X(n) USAGE IS DISPLAY.  
* n = length of FILE-NAME including terminating NULL or blank.  
77 NEWNAME PIC X(m) USAGE IS DISPLAY.  
* m = length of NEW-NAME including terminating NULL or blank.  
77 RET PIC S9(8) USAGE IS COMP-5.
```

```
CALL "ADX_CRENAME_FILE" USING RET,  
BY REFERENCE FILENAME, BY REFERENCE NEWNAME.
```

## Parameters:

**filename** Address of NULL or blank terminated string containing the current name of the file (Max length = 128 including NULL)

**Note:** When using the 32-bit API to access files on a drive enabled for long file name support, the file name is still considered to end at the first blank. The maximum length of a long file name is 260 bytes including the NULL or blank.

**newname** Address of NULL or blank terminated string containing the new name for the file (Max length = 128 including NULL)

**Note:** When using the 32-bit API to access files on a drive enabled for long file name support, the file name is still considered to end at the first blank. The maximum length of a long file name is 260 bytes including the NULL or blank.

**ret** Return code. It contains an error code if an error occurred.

See the *4690 OS: Messages Guide* for system errors.

## Seek (Change or Get File Pointer)

This operation either returns or changes the file pointer position for the specified file. To get the current pointer position, select the Relative to file pointer option in flag bits 8 and 9 and specify an offset of 0. Any other combination of values for flag bits 8 and 9 and the offset cause a change in the file pointer position. For all calls, a positive return value indicates the current file pointer position.

The offset value can be positive or negative. An error is returned, however, if the new pointer position is less than 0. If the file consists of multibyte records, the offset must fall on a record boundary.

### 16-Bit C Interface:

```
void ADX_CSEEK_PTR(long *pposit, unsigned int flags, long fnum, long boffset);
```

### 32-Bit C Interface:

```
void ADX_CSEEK_PTR(long *pposit, unsigned short flags, long fnum, long boffset);
```

### COBOL Interface:

77	PPOSIT	PIC S9(8)	USAGE IS COMP-5.
77	FLAGS	PIC 9(4)	USAGE IS COMP-5.
77	FNUM	PIC S9(8)	USAGE IS COMP-5.
77	BOFFSET	PIC S9(8)	USAGE IS COMP-5.

```
CALL "ADX_CSEEK_PTR" USING PPOSIT, BY VALUE FLAGS,  
BY VALUE FNUM, BY VALUE BOFFSET.
```

### Parameters:

#### flags

**bits 0 to 7:** Reserved (must be zero)

**bits 8 and 9:** Determine how the offset field is interpreted

- 00 = Relative to beginning of file
- 01 = Relative to file pointer
- 10 = Relative to end of file

**bits 10 to 15:** Reserved (must be zero)

**fnum** File number as returned from an OPEN or a CREATE.

**boffset** Number of bytes relative to reference selected in flag bits 8 and 9.

**pposit** Return code. It contains the current position of the file pointer after the call or an error code if an error occurred.

See the *4690 OS: Messages Guide* for system errors.

## Change File Attributes

This operation changes disk file attributes to enable a file to be distributed. To use this operation, the file must be open on the store controller with ownership of the file. The master controller owns compound files and the master file server owns mirrored files. After the attributes have been changed at least one record in the file must be written so that the operating system marks the file for distribution. If the file has never been distributed or if there is currently no copy on the other store controllers, the receiving store controllers must be IPLed for the file to be distributed.

### 16-Bit C Interface:

```
void ADX_CCHANGE_ATTRIB(long *ret, long fnum, unsigned int ftype,  
unsigned int fdistrib, long recsize);
```

### 32-Bit C Interface:

```
void ADX_CCHANGE_ATTRIB(long *ret, long fnum, unsigned short ftype,  
unsigned short fdistrib, long recsize);
```

### COBOL Interface:

```
77 FNUM          PIC S9(8)          USAGE IS COMP-5.  
77 FTYPE         PIC 9(4)           USAGE IS COMP-5.  
77 FDISTRIB      PIC 9(4)           USAGE IS COMP-5.  
77 RECSIZE       PIC 9(8)           USAGE IS COMP-5.  
77 RET           PIC S9(8)          USAGE IS COMP-5.
```

```
CALL "ADX_CCHANGE_ATTRIB" USING RET,  
BY VALUE FNUM, BY VALUE FTYPE, BY VALUE FDISTRIB,  
BY VALUE RECSIZE.
```

### Parameters:

**fnum** File number returned by a create or an open.

**ftype**

0 = Local only file  
1 = Mirrored file  
2 = Compound file

**fdistrib**

0 = Distribute at close  
1 = Distribute Per Update

**recsize** Size of record, as specified when the file was created

**ret** Return code. An error code is returned if an error occurred.

See the *4690 OS: Messages Guide* for system errors.

### Canceling the Shared Use of a File

ADX\_CFILES cancels the shared use of a file such as the transaction log. Without the use of ADX\_CFILES, a shared file cannot be renamed or deleted because it is in use by more than one user.

Some of the conditions that can cause file sharing problems are incomplete transactions, terminal hardware failures, incomplete controller functions, and software failures. ADX\_CFILES provides the following functions to manage the canceling of shared file usage:

- Restricting a file for exclusive use
- Unrestricting a file that was restricted
- Determining despool status

**ADX\_CFILES Restrict:** ADX\_CFILES restrict forces the exclusive use of a single file. This function is intended to be used only for store closing procedures and should not be used as a general purpose function.

ADX\_CFILES restrict function causes all applications currently using that file to lose their access to that file until they have closed and reopened the file. When an application attempts to use a file that has been restricted, the file request is rejected and a new return code is returned. In the controller the return code is 80F306F0. In the terminal, the first terminal access to a restricted file receives a 80F306F0 and subsequent accesses (by the same or other terminals) receive a 80004007 (bad file number). When an application has a file open and receives either of these return codes for a file request, it should close and reopen that file.

- Purpose:
 

To restrict the use of a file by all other applications. The restrict applies to applications on all controllers and all terminals. To restrict the use of a mirrored or compound file you restrict the prime version of the file on the file server or master respectively. You cannot use restrict for a distribute-on-close type file.

After a file is restricted:

  - It cannot be opened by any application.
  - An application that is using a file that is restricted by another application must close and reopen a file to use the file again.
- Restrictions:
  - Only one ADX\_CFILES to restrict a file can be active at a time.
  - To restrict a file on a file server or a master the controller application executing the restrict must be connected to the active file server or master.
  - To restrict a mirrored file or compound file an application must restrict the prime version of the file.
  - A distribute-on-close type file cannot be restricted.
- Location of usage:
  - Any application on any controller.
  - It cannot be used by a terminal application.

#### **16-Bit C Interface:**

```
void ADX_CFILES(long *ret, unsigned int func, char *filename);
```

#### **32-Bit C Interface:**

```
void ADX_CFILES(long *ret, unsigned short func, char *filename);
```

#### **COBOL Interface:**

This function is currently not supported for COBOL.

#### **Parameters:**

<b>ret</b>	= 4-byte integer that indicates the status of the restrict function
<b>hi lo</b>	
<b>xx xx yy yy</b>	= Indicates how the file was being used when the restrict was executed:
<b>xxxx</b>	= Number of other controller applications that were using this file when it was restricted.
<b>yyyy</b>	= Number of controllers where terminal applications were using this file when it was restricted.
<b>0000</b>	= No other application has the file open.
<b>80 F3 06 F4</b>	= Application attempted to restrict a file while another restrict is active.
<b>80 F3 06 F5</b>	= Application attempted to restrict a distribute-on-close file.
<b>80 F3 06 F6</b>	= Application attempted to restrict the file on the wrong node.
<b>8x xx xx xx</b>	= Any other operating system error return code.
<b>-1201</b>	= Invalid function code used for the FUNC values defined for ADX_CFILES.
<b>-1203</b>	= FILENAME is not defined.
<b>-1204</b>	= FILENAME is not a valid file name.
<b>-1205</b>	= Force Close support is not installed.
<b>func</b>	= 1 (This is the restrict function code for ADX_CFILES.)
<b>filename</b>	= String containing the file name of the file to be restricted. The name can be a logical name or a fully-qualified file name. To reference a mirrored or compound file use the generic node names for the file server and the master: <ul style="list-style-type: none"> <li>• for file server use ADXLXACN::</li> <li>• for master use ADXLXAAN::</li> </ul>

**Note:** When an application attempts to use a file that has been restricted, you receive either an 80F306F0 or an 80004007. The error recovery should provide a delay and retry mechanism because the file remains restricted until it is unrestricted by the application.

**ADX\_CFILES Unrestrict:** ADX\_CFILES unrestrict stops the effect of the ADX\_CFILES restrict. The unrestrict is requested for the same file name that was used for the restrict even if the file was renamed while it was restricted. After the unrestrict is executed that file name can be used by all applications again.

- Purpose:

To unrestrict the use of a file that had been previously restricted. To unrestrict the use of a Mirrored or Compound file, unrestrict the prime version of the file on the File Server or Master respectively. After a file is unrestricted:

- The file name can be used to open files according to the normal guidelines.
- An application that lost access to a file due to restrict must issue a CLOSE followed by an OPEN after the unrestrict is issued to gain access to the file again.

- Prerequisites:

The application executing the unrestrict ADX\_CFILES must have executed the restrict ADX\_CFILES.

- Restrictions:

If an application is unrestricting a file on a file server or a master, the controller executing the application must be connected to the active file server or master.

- Location of usage:

- Any application on any controller.
- It cannot be used by a terminal application.

### 16-Bit C Interface:

```
void ADX_CFILES(long *ret, unsigned int func, char *filename);
```

### 32-Bit C Interface:

```
void ADX_CFILES(long *ret, unsigned short func, char *filename);
```

### COBOL Interface:

This function is currently not supported for COBOL.

### Parameters:

**ret** = 4-byte integer that identifies the unrestrict status.

hi	lo	
00 00	00 00	= Unrestrict is successful.
80 F3	06 F2	= Application attempted to do an unrestrict without doing a restrict.
80 F3	06 F3	= Application attempted to unrestrict a file that it did not have restricted.
8x xx	xx xx	= Any other OS error return code.
-1201		= Invalid function code.
-1203		= FILENAME is not defined.
-1204		= FILENAME is not a valid file name.
-1205		= Force Close support is not installed.

When an unsuccessful unrestrict is executed because of a LAN failure (80 60 xx xx), the application should wait 30 seconds and retry the unrestrict. This should be repeated for a total of 7 executions of unrestrict.

**func** = 2 (This is the unrestrict function code for ADX\_CFILES.)

**filename** = String containing the file name of the file to be restricted. The name can be a logical name or a fully qualified file name. To reference a mirrored or compound file use the generic node names for the file server and the master store controller:

- For file server use ADXLXACN::
- For master use ADXLXAAN::

**ADX\_CFILES Despool:** ADX\_CFILES despool determines how many total bytes of data remain in all the spool files and how many controllers have spool files. By using this function the applications can determine that the store personnel should be notified that a store closing must be delayed and can give the store personnel an idea of the length of the delay.

- Purpose:

This function determines the status of the operating system despooling of files. This function determines how many bytes of data are yet to be despoiled and how many controllers have data in their spool files to be despoiled. The values determined by this function are probably different than the sizes of the spool files because the size of the spool files are not set to zero until all the data has been despoiled from them.

- Prerequisites:

None currently identified.

- Restrictions:

This function can only determine despool status for controllers that are currently in session with this controller on the LAN.

- Location of Usage:

- Any application on any controller.
- It cannot be used by a terminal application.

**16-Bit C Interface:**

```
void ADX_CFILES(long *ret, unsigned int func, char *filename);
```

**32-Bit C Interface:**

```
void ADX_CFILES(long *ret, unsigned short func, char *filename);
```

**COBOL Interface:**

This function is currently not supported for COBOL.

**Parameters:**

**ret** = 4-byte integer that specifies the despool status.

**hi lo**

**0w xx yy yy** = where:

**w** = Status of LAN communications.

**0** = All controllers are communicating with this controller.

**8** = One or more controllers are not communicating with this controller.

**xx** = Number of communicating controllers with data to despool.

**yyyy** = Total number of bytes (in 1000s) of data to be despoiled by controllers communicating.

**00 00 00 00** = The system supports LAN and there is no despooling to do, or the system does not support LAN.

**8x xx xx xx** = Indicates any operating system error return code:

**-1201** = Function code is not valid.

**-1205** = System supports LAN but Force Close support is not installed.

**func** = 3 This is the despool function code for ADX\_CFILES.

**filename** = Unused; the value is ignored.

## Pipe Routing Services

### Create a Pipe Routing Services Pipe

Pipe Routing Services pipes are created for exclusive, READ mode only.

#### 16-Bit C Interface:

```
void ADX_CPRS_CREATE(long *pnum, long psize, char *pipeid);
```

#### 32-Bit C Interface:

```
void ADX_CPRS_CREATE(long *pnum, long psize, char *pipeid);
```

#### COBOL Interface:

77	PNUM	PIC S9(8)	USAGE IS COMP-5.
77	PSIZE	PIC 9(8)	USAGE IS COMP-5.
77	PIPEID	PIC X	USAGE IS DISPLAY.

```
CALL "ADX_CPRS_CREATE" USING PNUM,  
BY VALUE PSIZE, BY REFERENCE PIPEID.
```

#### Parameters:

<b>psize</b>	Pipe size, in bytes. The maximum size for store controller pipe is 65 536 bytes.
<b>pipeid</b>	One alphabetic character (A to Z) to identify the pipe.
<b>pnum</b>	Return code. It contains the pipe identifier or an error code if the create was unsuccessful.

The unique error codes are “-1000 Invalid pipe id” and “-1001 Pipe already exists.” See the *4690 OS: Messages Guide* for system errors.

### Read from a Pipe Routing Services Pipe

#### 16-Bit C Interface:

```
void ADX_CPRS_READ(long *nbytes, long pnum, char *buffadr, long buffsize);
```

#### 32-Bit C Interface:

```
void ADX_CPRS_READ(long *nbytes, long pnum, char *buffadr, long buffsize);
```

#### COBOL Interface:

77	NBYTES	PIC S9(8)	USAGE IS COMP-5.
77	PNUM	PIC S9(8)	USAGE IS COMP-5.
77	BUFFADR		USAGE IS POINTER.
77	BUFFSIZE	PIC 9(8)	USAGE IS COMP-5.

```
CALL "ADX_CPRS_READ" USING NBYTES,  
BY VALUE PNUM, BY VALUE BUFFADR, BY VALUE BUFFSIZE.
```

#### Parameters:

<b>pnum</b>	Pipe identifier returned from pipe routing services CREATE.
<b>buffadr</b>	Buffer to receive data
<b>buffsize</b>	Number of bytes to read
<b>nbytes</b>	Return code. It contains the number of bytes read or an error code if an error occurred.

See the *4690 OS: Messages Guide* for system errors.

### Initialize Pipe Routing Service Driver

The Pipe Routing Services Driver must be initialized before an application can write to a pipe routing services pipe. It should be initialized only once for each load of an application.

### 16-Bit C Interface:

```
void ADX_CPRS_INIT(long *prsnm);
```

### 32-Bit C Interface:

```
void ADX_CPRS_INIT(long *prsnm);
```

### COBOL Interface:

```
77 PRSNM          PIC S9(8)          USAGE IS COMP-5.

CALL "ADX_CPRS_INIT" USING PRSNM.
```

### Parameters:

**prsnm** Return code. It contains the PRS identifier or an error code if an error occurred.

See the *4690 OS: Messages Guide* for system errors.

## Write to a Pipe Routing Services Pipe

### 16-Bit C Interface:

```
void ADX_CPRS_WRITE(long *ret, long prsnm, char *buffadr, long buffsize,
char *dest);
```

### 32-Bit C Interface:

```
void ADX_CPRS_WRITE(long *ret, long prsnm, char *buffadr, long buffsize,
char *dest);
```

### COBOL Interface:

```
77 RET            PIC S9(8)          USAGE IS COMP-5.
77 PRSNM          PIC S9(8)          USAGE IS COMP-5.
77 BUFFADR        PIC X(4)           USAGE IS POINTER.
77 BUFFSIZE       PIC 9(8)           USAGE IS COMP-5.
77 DEST           PIC X(4)           USAGE IS DISPLAY.
```

```
CALL "ADX_CPRS_WRITE" USING RET,
BY VALUE PRSNM, BY VALUE BUFFADR, BY VALUE BUFFSIZE,
BY REFERENCE DEST.
```

### Parameters:

**prsnm** PRS identifier returned from ADX\_CPRS\_INIT

**buffadr** Buffer containing data to write

**buffsize** Number of bytes to write

**dest** Destination address (where to send data). It contains four characters in the form *aaaw*. The *aaaw* indicates which terminal or store controller to send the data to. The *w* part of the destination address is the pipeID for a pipe routing services pipe created by an application executing in the specified store controller or terminal. For terminals, *aaa* is the terminal number in ASCII. For store controllers, it is *Oxy* where *O* is an ASCII zero, *x* and *y* are ASCII characters between C and Z. There are two special values for *Oxy* for store controllers: *OAA* and *OBB*. Use *OAA* when the destination is the master store controller and use *OBB* when the destination is the store controller where the calling application is executing (in the local store controller). Pipe routing services translates these special destination addresses so that the application does not need to define the actual address assigned to the physical machine. If the operating system switches destinations when a configuration changes, PRS translates the change and no change is required for the application because of the switch.



**ret** Return code. Table 42 on page 363 shows the return codes for the Pipe Routing Services function.

Table 42. Pipe Routing Services Return Codes

Return Code	Description
0	Good completion.
-1010	Invalid destination specified.
-1011	Destination not found.
-1012	Error on write to destination.
-1013	Data greater than 120-byte maximum.

## Conditional Write to a Pipe Routing Services Pipe

### 16-Bit C Interface:

```
void ADX_CPRS_CWRITE(long *ret, long prsnum, char *buffadr, long buffsize,
char *dest);
```

### 32-Bit C Interface:

```
void ADX_CPRS_CWRITE(long *ret, long prsnum, char *buffadr, long buffsize,
char *dest);
```

### COBOL Interface:

```
77 RET          PIC S9(8)          USAGE IS COMP-5.
77 PRSNUM       PIC S9(8)          USAGE IS COMP-5.
77 BUFFADR      PIC S9(8)          USAGE IS POINTER.
77 BUFFSIZE     PIC 9(8)           USAGE IS COMP-5.
77 DEST        PIC X(4)           USAGE IS DISPLAY.
```

```
CALL "ADX_CPRS_CWRITE" USING RET,
BY VALUE PRSNUM, BY VALUE BUFFADR, BY VALUE BUFFSIZE,
BY REFERENCE DEST.
```

### Parameters:

**prsnum** PRS identifier returned from ADX\_CPRS\_INIT

**buffadr** Buffer containing data to write

**buffsize** Number of bytes to write

**dest** Destination address (where to send data). It contains four characters in the form *aaaw*. The *aaaw* indicates the terminal or store controller to which send the data. The *w* part of the destination address is the pipe ID for a pipe routing services pipe created by an application executing in the specified store controller or terminal. For terminals, *aaa* is the terminal number in ASCII. For store controllers, it is *0xy* where *0* is an ASCII zero, *x* and *y* are ASCII characters between C and Z. There are two special values for *0xy* for store controllers: *0AA* and *0BB*. Use *0AA* when the destination is the master store controller and use *0BB* when the destination is the store controller where the calling application is executing (in the local store controller). Pipe routing services translates these special destination addresses so that the application does not need to define the actual address assigned to the physical machine. If the operating system switches destinations when a configuration changes, PRS translates the change and no change is required for the application because of the switch.

**ret** Return code. Table 43 on page 364 contains return codes for the Conditional Write to Pipe Routing Services Pipe function.

Table 43. Return Codes for Conditional Write to a Pipe Routing Services Pipe

Return Code	Description
0	Good completion occurred.
-1	Destination pipe is full or does not have enough space available in the pipe to hold the data written.
-1010	The specified destination specified is not valid.
-1011	Destination is not found.
-1012	Error on write to destination.
-1013	Data greater than 120-byte maximum

**Usage:** This write is similar to ADX\_CPRS\_WRITE with the following exception: If the destination pipe is full or does not have enough room left to contain the entire message written, the write does not wait for room to become available. Instead, the application is given control immediately with a -1 return code. At this point the application must make a decision to either discard the data being written or retry the write at a later time.

The intended use for the conditional pipe write is for situations where it is undesirable for an application to wait for an extended period of time.

## Communications Services

The following application interfaces are described in the *4690 OS: Communications Programming Reference*:

<b>ADX_COPEN_LINK</b>	Open communications link
<b>ADX_COPEN_SESS</b>	Open session
<b>ADX_CCLOSE_LS</b>	Close communications link or session
<b>ADX_CREAD_HOST</b>	Read data from link or session
<b>ADX_CWRITE_HOST</b>	Write data to link or session
<b>ADX_CGET_STAT</b>	Obtain status from link or session
<b>ADX_CSEND_REQ</b>	Requests to the driver

## Specialized Services

The Specialized Services function provide a variety of functions for controller applications written in C language and COBOL. For background information, see Chapter 16, "Designing applications with 4680 BASIC," on page 277 for a description of these services for BASIC applications.

### Operator Authorization

This authorization function can be used to create and maintain authorization records that enable operators to sign on and use the system.

This authorization function is only valid for store controller applications written in C language and COBOL. If the authorization function requested is change or add, the user is prompted for input. When the add or change is complete, the screen is cleared and the cursor is enabled and returned to the home position.

#### 16-Bit C Interface:

```
void ADX_CAUTH(long *ret, int func, char *opid, char *password, char *opid2);
```

#### 32-Bit C Interface:

```
void ADX_CAUTH(long *ret, short func, char *opid, char *password, char *opid2);
```

### COBOL Interface:

```
77 RET          PIC S9(8)          USAGE IS COMP-5.
77 FUNC         PIC 9(4)           USAGE IS COMP-5.
77 OPID         PIC X(9)           USAGE IS DISPLAY.
77 PASSWORD     PIC X(8)           USAGE IS DISPLAY.
77 OPID2        PIC X(19)          USAGE IS DISPLAY.
```

```
CALL "ADX_CAUTH" USING RET, BY VALUE FUNC,
BY REFERENCE OPID, BY REFERENCE PASSWORD, BY REFERENCE OPID2.
```

### Parameters:

<b>func</b>	Action to be taken: <b>1</b> = CHANGE an operator authorization record <b>2</b> = ADD an operator authorization record <b>3</b> = DELETE an operator authorization record <b>8</b> = CHANGE PASSWORD only <b>9</b> = MAKE operator ID have the same authorization as the operator ID specified by opid2. If opid2 does not exist, a new ID is created with limited (default) authorization. <b>10</b> = MAKE WITH CHECK — same as 9 except opid2 has template ID and authority ID. Authorization for new ID cannot be higher than authority ID even if the template authorization is higher.
<b>opid</b>	Operator ID on which action is to be taken. This ID should be an ASCII string of up to nine characters. If the ID is less than nine characters it must be terminated with a blank or a NULL. This parameter should have no leading blanks. Leading blanks and zeros are counted as part of the ID.
<b>password</b>	Password for ID. The password is an ASCII string of up to eight characters. If the string is less than eight characters it must be terminated with a blank or NULL. This parameter should have no leading blanks. Leading zeros are counted as part of the password.
<b>opid2</b>	Current operator ID for functions 1, 2, 3, and 8 or template ID for functions 9 and 10, depending on action requested. This ID can be up to nine ASCII characters and it must be terminated with a blank or NULL if it is less than nine characters. However, if the action to be taken is MAKE WITH CHECK, opid2 must have the template ID of up to nine characters, then a colon, and then the current operator ID of up to nine characters terminated with a blank or NULL if either ID is less than nine characters. There should be no leading or imbedded blanks. Leading blanks and zeros are counted as part of the ID.
<b>ret</b>	Return code. It contains a negative integer if an error occurred. Refer to Table 44 on page 366 for a list of error codes.

**Note:** At least one ID on every system should be authorized for all system functions. The default authorization allows the operator to sign on and select only primary or secondary applications. The current operator ID making a change to a record cannot be the same as the operator ID being changed.

- If the action to be taken is ADD or CHANGE, the system displays several panels to select the system functions that can be used. The current operator (opid2) can select only those functions for which the current operator ID is authorized.
- If the action to be taken is a CHANGE for an ID that does not exist, the ID is added using default authorization and error code -1010 is returned.
- If the action to be taken is an ADD for an ID that already exists, the action is handled as a CHANGE and error code -1010 is returned.
- If the action to be taken is MAKE or MAKE WITH CHECK, an ID can be added using a template in opid2 without having to select each system function from the various panels. The authorization allowed is the same as the template. For a MAKE WITH CHECK, the authorization only includes functions for

which both the template ID and the current operator ID have authorization. If opid2 is missing, or for MAKE WITH CHECK if either part is missing, error code -1011 is returned and the default authorization is used as the template.

Table 44 on page 366 shows the return codes returned by this function.

Table 44. Function Return Codes

Error Code	Description
0	Function completed successfully.
-1000	Unknown function code.
-1001	File not found.
-1002	Error reading or writing the disk.
-1003	OPID missing or not valid.
-1004	Password missing or not valid.
-1005	OPID2 missing or not valid.
-1010	One of these conditions exists: <ul style="list-style-type: none"> <li>• Add function was handled as change because ID already exists.</li> <li>• Change function was handled as an add because ID was not found.</li> <li>• Delete requested for ID that was not found.</li> </ul>
-1011	Function handled as requested using default authorization because OPID2 not found.
-1020	Could not allocate memory for ID search.

For any system errors that are returned, see the *4690 OS: Messages Guide*.

## Password Encryption

This function encrypts an eight-character ASCII password. The operating system uses one-way encrypted passwords for authorization to sign on to the system. The encryption method can also be used by an application to establish authorization for use of applications, data files, or other things that need access controlled by the application.

### 16-Bit C Interface:

```
void ADX_CCRIPT(long *ret, char *pwin, char *pwout);
```

### 32-Bit C Interface:

```
void ADX_CCRIPT(long *ret, char *pwin, char *pwout);
```

### COBOL Interface:

```

77 RET          PIC S9(8)          USAGE IS COMP-5.
77 PWIN         PIC X(8)           USAGE IS DISPLAY.
77 PWOUT        PIC X(8)           USAGE IS DISPLAY.

CALL "ADX_CCRIPT" USING RET, PWIN, PWOUT.
```

### Parameters:

**pwin** Password to be encrypted. Must be terminated with a NULL or blank if less than eight characters.

**pwout** Encrypted value returned.

**ret** Return code = 0 indicates encryption completed. If PWIN is blank or has a leading blank, error code -1100 is returned.

The password to be encrypted should be an ASCII string of up to eight alphanumeric characters. This parameter can have no leading blanks. Trailing blanks are ignored. Leading zeros are counted as part of the password. The encrypted value returned in PWOUT is an eight-character string of ASCII characters that represents decimal digits.

## Common Application Services (ADX\_CSERVE)

This function can be called from a store controller application to request one of the application services.

The parameters here are slightly different from those described for ADXSERVE.

### 16-Bit C Interface:

```
void ADX_CSERVE(long *ret, int funcnum, char *databuff, int dbuffsize);
```

### 32-Bit C Interface:

```
void ADX_CSERVE(long *ret, short funcnum, char *databuff, short dbuffsize);
```

### COBOL Interface:

77	RET	PIC S9(8)	USAGE IS COMP-5.
77	FUNCNUM	PIC 9(4)	USAGE IS COMP-5.
77	DATABUFF		USAGE IS POINTER.
77	DBUFFSIZE	PIC 9(4)	USAGE IS COMP-5.

```
CALL "ADX_CSERVE" USING RET, BY VALUE FUNC-NUM,  
BY VALUE DATA-BUFF, BY VALUE DBUFF-SIZE.
```

### Parameters:

<b>funcnum</b>	Function number for the service requested.
<b>databuff</b>	Buffer for data sent to service requested or for data received from service requested.
<b>dbuffsize</b>	Size (bytes) of data-buff or data for requested service if data-buff is not used.
<b>ret</b>	Return code values vary with the service requested. An error code is returned if an error occurred.

Special error codes are returned. See Table 31 on page 294 for a list of the return codes. For any system errors that are returned, see the *4690 OS: Messages Guide*.

## Application Services for C-language and COBOL

The following is a list of the Application Services available when ADX\_CSERVE is called. The function number and the required data, if any, are listed for each service.

**Dump System Storage:** The Dump System Storage function causes all of the storage in the controller to be dumped to a disk file named ADXCSLCF.DAT in the root directory. Both the application and the operating system storage are dumped.

This function uses the following parameters:

<b>FUNC-NUM</b>	= 1
<b>DATA-BUFF</b>	= Unused
<b>DBUFF-SIZE</b>	= Unused

**Enable Terminal Storage Retention:** This function enables storage retention in all of the terminals on the TCC Network of the store controller requesting the enable.

The Enable Terminal Storage Retention function uses the following parameters:

<b>FUNC-NUM</b>	= 2
<b>DATA-BUFF</b>	= Unused
<b>DBUFF-SIZE</b>	= Unused

**Disable Terminal Storage Retention:** This function disables storage retention in all of the terminals on the TCC Network of the store controller requesting the disable.

The Disable Terminal Storage Retention function uses the following parameters:

**FUNC-NUM** = 3  
**DATA-BUFF** = Unused  
**DBUFF-SIZE** = Unused

**Get Application Status Information:** The Get Application Status Information function returns the store controller application status. See “Get Application Status Information” on page 296 for the format of the data returned.

This function uses the following parameters:

**FUNC-NUM** = 4  
**DATA-BUFF** = Address of buffer to receive status information  
**DBUFF-SIZE** = 80 bytes

**Programmable Power:** The programmable power function enables terminal or controller applications to issue program calls to enable or disable the programmable power feature, and to issue program calls to power off a terminal or controller.

**Note:** Programmable power is not supported on the terminal side of a controller/terminal.

This function uses the following parameters:

**FUNC-NUM** 5  
**DATA-BUFF** Eight-byte buffer containing:

- Two-byte integer set to 5 (for function 5)
- Two-byte integer specifying subfunction. Valid values are 1, 2, 3, 4, or 5. These values correspond to PARM5 values on ADXSERVE.
- Four-byte pointer to a character string. The character string pointed to by this pointer can have these valid formats:
  - *DDHHMM* where:
    - DD** = The day of the month (01-31).
    - HH** = The hours of the day (00-24).
    - MM** = The minutes of the hour (00-59).
  - *DDHHMMTTT* where:
    - DD** = The day of the month (01-31).
    - HH** = The hours of the day (00-24).
    - MM** = The minutes of the hour (00-59).
    - TTT** = The terminal number (001-999).
  - *DDHHMMCC* where:
    - DD** = The day of the month (01-31).
    - HH** = The hours of the day (00-24).
    - MM** = The minutes of the hour (00-59).
    - CC** = The controller ID (CC through ZZ).

**DBUFF-SIZE** = Size of the DATA-BUFF (eight bytes).

**Restart a background application in the same slot:** This function allows a configured background application which has ended to be restarted in the same slot it previously occupied. This service can be used to stop a background application as well.

**Note:** The name field is not case sensitive, parameters are case sensitive.

The program is responsible for checking user access to this service. This API will only work locally, not remotely from one controller to another.

This function uses the following parameters:

**FUNC-NUM** = 19  
**DATA-BUFF** = String of data specifying the application name (1 to 24 characters) and optionally the parameter string (1 to 45 characters). If there is a parameter string, the application name must be padded with blanks allowing 24 characters to be present prior to the parameter data, if needed. The parameter field would then be from 1 to 45 characters.  
**DBUFF-SIZE** Specify one of the following:  
1 = To start a background application.  
2 = To start a background application.

**Display Application Status Message:** The Display Application Status Message function displays the specified message on the WINDOW CONTROL panel in the description field of the window for the application that called this function.

This function uses the following parameters:

**FUNC-NUM** = 25  
**DATA-BUFF** = Address of buffer containing message to display  
**DBUFF-SIZE** = Size (bytes) of message, (maximum = 79 bytes.)

**Display Background Application Status Message:** The Display Background Application Status Message function displays the specified message on the BACKGROUND APPLICATION CONTROL panel in the message field of the requesting background application. The message is displayed until the message is changed or the application ends.

This function uses the following parameters:

**FUNC-NUM** = 26  
**DATA-BUFF** = Address of buffer containing message to display  
**DBUFF-SIZE** = Size (bytes) of message, (maximum = 46 bytes.)

**Stop Application with Message:** The Stop Application with Message function displays the specified message on the WINDOW CONTROL panel used to start the application. The message appears after the application is stopped. This function provides a way to indicate problems that are preventing an application from running properly.

This function uses the following parameters:

**FUNC-NUM** = 27  
**DATA-BUFF** = Address of buffer containing message to display  
**DBUFF-SIZE** = Size (bytes) of message. Maximum = 79 bytes.

**Get Disk Free Space:** The Get Disk Free Space function returns the free space on the specified remote or local disk drive. Example: local = C:\, non-local = ADXLNXnnN::C:\ (where nn = store controller ID). Count of characters should be exact, not general size of DATA-BUFF.

The free space is returned in the RET variable. This value defaults to bytes, and the maximum value that can be returned is (decimal) 2147483647, which is 2GB - 1. The free space can be returned in kilobytes by including 1024 in the DATA-BUFF string variable. This allows the actual free space to be returned when that value is greater than 2GB. For example, if DATA-BUFF is C:\ 1024, the function returns the free space on the C: drive in kilobytes.

This function uses the following parameters:

**FUNC-NUM** = 28  
**DATA-BUFF** = Address of buffer containing the node name (if non-local) and the drive, or a logical name for the node and/or the drive  
**DBUFF-SIZE** = Number of characters in name. Maximum = 127 bytes.



**Get Configured Controllers on the Network:** This function returns the IDs of all the store controllers on the network. Each ID is two bytes long. Store controller IDs range from CC through ZZ. There can be up to eight controllers configured. If there are fewer than eight, an ID of 00 (ASCII zeroes, not numeric) indicates the end of the list.

This function uses the following parameters:

<b>FUNC-NUM</b>	= 29
<b>DATA-BUFF</b>	= Address of buffer to receive the list of store controllers.
<b>DBUFF-SIZE</b>	= 40 bytes

**Get The Controller ID for a Specified Terminal:** The Get The Controller ID for a Specified Terminal function returns the store controller ID for the specified terminal. The ID is returned in the RET parameter as ASCII value CC through ZZ or X'0' if the terminal was not defined. These values are returned only if the store controller requesting the ID is the master store controller or if the terminal is local to the store controller.

This function uses the following parameters:

<b>FUNC-NUM</b>	= 30
<b>DATA-BUFF</b>	= Unused
<b>DBUFF-SIZE</b>	= Terminal number for which the ID is requested

**Convert ASCII Characters to EBCDIC Characters:** This function converts ASCII characters to EBCDIC characters. The characters are changed byte by byte, therefore the original characters are no longer present when the function completes the conversion.

This function uses the following parameters:

<b>FUNC-NUM</b>	= 31
<b>DATA-BUFF</b>	= Address of buffer containing ASCII characters to convert
<b>DBUFF-SIZE</b>	= Number of characters to convert

**Convert EBCDIC Characters To ASCII Characters:** This function converts EBCDIC characters to ASCII characters. The characters are changed byte by byte, therefore the original characters are no longer present when the function completes the conversion.

The function uses the following parameters:

<b>FUNC-NUM</b>	= 32
<b>DATA-BUFF</b>	= Address of buffer containing EBCDIC characters to convert
<b>DBUFF-SIZE</b>	= Number of characters to convert

**Set/Reset/Query Terminal Dump Preservation Flag:** This function lets you set, reset, and query the terminal dump preservation flag. Setting the preservation flag prevents a terminal dump from being written to the terminal dump file. Resetting the preservation flag allows a dump to be written to the file. The query request returns the status of the preservation flag.

A return code of 1 indicates that the preservation flag is ON. A return code of 0 indicates that the preservation flag is OFF.

If the user logical name ADXTDUMP has not been created to enable the Terminal Dump Preservation Function, you receive a return code of -1022 if you try to access this function.

This function uses the following parameters:

<b>FUNC-NUM</b>	= 33
<b>DATA-BUFF</b>	= Unused
<b>DBUFF-SIZE</b>	= Specify one of the following:
<b>0</b>	To turn off the terminal dump preservation flag
<b>1</b>	To Turn on the terminal dump preservation flag



- 2 To query whether the terminal dump preservation flag is ON or OFF

**Get Loop Message:** This function gets the three most recent system messages for the token-ring adapter specified in DATA-BUFF. DATA-BUFF is a string consisting of a node (for example, CD), a TCC Network (1 or 2), and three TCC Network messages.

The node and the TCC Network must be the first three characters of the string when the ADX\_CSERVE function is called. When the ADX\_CSERVE function returns, the three most recent messages follow the node and TCC Network in the string. If no messages exist for the specified node and TCC Network, blanks are returned for the messages. The oldest message is put in the buffer first. Each message is 133 characters long.

The function uses the following parameters:

**FUNC** = 34  
**DATA-BUFF** = Address of the buffer  
**DBUFF-SIZE** = 402 bytes

**Get Loop Status:**

**Note:** Store Loop support was removed in 4690 OS V6R3.

This function returns the configuration and current status of the two loop adapters. Data is returned in RET in the form WWXXYYZZ (hex) where:

**ZZ** = First adapter configuration  
**YY** = First adapter status  
**XX** = Second adapter configuration  
**WW** = Second adapter status

The configuration is defined as:

**00** = Not used  
**01** = Primary  
**02** = Secondary  
**04** = 2400 bps loop  
**80** = Auto resume of primary loop control

The status is defined as:

**03** = Standby  
**04** = Controlling  
**05** = Prevented

The function uses the following parameters:

**FUNC-NUM** = 35  
**DATA-BUFF** = Unused; the value is ignored.  
**DBUFF-SIZE** = Unused; the value is ignored.

**Get All Active Controllers on the Network:** This function returns the IDs of all active store controllers on the network. Each ID is two bytes long. A store controller ID of 00 indicates the end of the list.

The function uses the following parameters:

**FUNC-NUM** = 36  
**DATA-BUFF** = Address of buffer to receive IDs  
**DBUFF-SIZE** = 40 bytes

### ***Get Controller ID and Loop for a Specified Terminal:***

**Note:** Store Loop support was removed in 4690 OS V6R3.

The data returned in RET is two bytes for the loop ID followed by the two bytes for the store controller ID. A X'01' is returned if the terminal number cannot be found.

**Note:** The RET is valid only if this function is issued at the master store controller.

The function uses the following parameters:

**FUNC-NUM** = 37  
**DATA-BUFF** = Address of a two-byte integer containing the terminal number  
**DBUFF-SIZE** = 2 bytes

***Get the Controller Machine Model and Type:*** This function returns the store controller model number and type. The function uses the following parameters:

**FUNC-NUM** = 38  
**DATA-BUFF** = Eight-byte buffer containing:

- 2-byte integer set to 38 (for function 38)
- 2-byte integer specifying subfunction 1. The valid value is 1.
- 4-byte pointer to a 4-byte buffer. The buffer pointed to by this pointer should have a length of four bytes. It contains the machine model and type. For example, for a 4693-541, the values returned are X'F8' X'3E' X'00' X'00'.

**DBUFF-SIZE** Size of the DATA-BUFF (8 bytes).

***Check the Master or File Server Exception Log:*** This function returns whether or not there are any entries in the exception log.

This function uses the following parameters:

**RET** = One of the following values:

- 0 = No entries in the exception log
- 1 = Entries exist in the exception log
- 8xxxxxxx = Error code indicating why the status of the exception log can not be obtained.

**FUNC** = 39  
**DATA-BUFF** Specify one of the following values:

- 1 = Check the Master exception log
- 2 = Check the File Server exception log

**DBUFF-SIZE** = Unused

***Set Terminal Sleep Mode Inactive Timeout:*** This function enables an application running in a store controller to set the Terminal Sleep Mode Inactive Timeout. When you enable Terminal Storage Retention, you set this value to determine how many minutes a terminal remains idle before being powered-down.

**Note:** This function is supported only on the store controller. In order for the sleep mode inactive timeout to take effect, you must invoke this function before enabling the Terminal Storage Retention. You can also specify the terminal sleep mode inactive timeout by selecting the Enable Terminal Storage Retention option on the TERMINAL FUNCTIONS panel. The default for the terminal sleep mode inactive timeout is 30 minutes.

This function uses the following parameters:

**FUNC-NUM** = 41  
**DATA-BUFF** = 2-byte buffer containing a two-byte integer specifying the terminal sleep

mode inactive timeout. Valid values are 0 through 255. You can use a 0 value to disable the terminal sleep mode.  
= Size of the DATA-BUFF (2 bytes).

#### DBUFF-SIZE

**Set Date and Time:** This function enables an application to set the system-wide date and time. Changes made via this function are broadcast to all controllers and all terminals within the store system.

This function uses the following parameters:

**FUNC-NUM** = 44  
**DATA-BUFF** = pointer to one of the following string commands:  
+ = increment the current time by 1 hour  
- = decrement the current time by 1 hour  
B = only broadcast current time to controllers and terminals  
HHMMSS = Hour Minute Second notation for the new current time  
HHMMSS YYMMDD = new Hour Minute Second and Year Month Date  
**DBUFF-SIZE** = Length of the string variable at DATA-BUFF.

**Note:** Whenever possible, the increment and decrement commands should be used over the set absolute commands because there is no way to guarantee with accuracy when the command is run. This function only runs on the master controller if on a multiple controller system. All of these commands broadcast their changes; you do not need to manually broadcast after making a change.

**Load Specific Terminal:** This function enables an application running in a store controller to load a specific terminal.

This function uses the following parameters:

**FUNC-NUM** = 42  
**DATA-BUFF** = Unused.  
**DBUFF-SIZE** = Terminal number to be loaded.

**Run Remote STC in One Terminal:** This function causes Remote Set Terminal Characteristics (STC) to be started in the terminal identified by the terminal number passed in the data buffer. The terminal number must be a valid terminal number and the terminal should be in communication with a controller. If the application using this ADXSERV call is running on the master controller, the terminal can be in communication with either the master controller or the backup controller. A return code, usually zero, is returned indicating that the Remote STC request has been delivered to the terminal. The -1002 return code indicates the terminal number is not a valid terminal number. The impact of this call is identical to the action requested from the terminal system control screen, option 9, where the user can request Remote STC be run on a particular terminal.

This function uses the following parameters:

**FUNC-NUM** = 47  
**DATA-BUFF** = Identifies the terminal number that Remote STC is to be run on.  
**DBUFF-SIZE** = Unused, value is ignored

**Run STC in All Terminals:** This function causes Remote Set Terminal Characteristics (STC) to be started in all terminals. The application using this API must be run on the master controller so all terminals in communication with any controller on the system has Remote STC run on them. No return codes are returned from this call since a broadcast message is delivered to the terminals that are currently listening. The impact of this call is identical to the action requested from the terminal system control screen, option 9 with \*, where the user can request Remote STC to be run on all terminals.

This function uses the following parameters:

**FUNC-NUM** = 48  
**DATA-BUFF** = Unused  
**DBUFF-SIZE** = Unused

**System Date (4-Digit Year) (ADXDATE):** The ADXDATE function writes the system date in the format YYYYMMDD into the caller's buffer. The buffer must be a minimum of nine characters long to accomodate the strign and a null terminator.

**16-Bit C Interface:** long adxdate(char \*buffer);

**32-Bit C Interface:** long adxdate(char \*buffer);

ADXDATE() returns 0 for successful completion. 1 is returned if the buffer argument is a null pointer or if an operating system error has occurred.

For controller applications, this function is in executable ADXAPACL.L86

**Retrieve Master Exception Logs:** This function uses the following parameters:

**RET**

**0** No entries in the Master Exception Logs  
**>0** Specifies the number of entries in the Master Exception Logs  
**xxxxxx** An error code indicating why the information cannot be obtained  
**FUNC-NUM** = 209  
**DATA-BUFF** = Buffer to retrieve Master Exception Logs  
**DBUFF-SIZE** = Size of DATABUFF

Upon a successful return, this will be in the user buffer, which has a minimum size of 87 bytes and a maximum size of 65424 bytes:

- Two bytes containing an integer value that specifies the number of missing entries
  - 0** The user buffer contains all entries that were found.
  - >0** The user buffer was not large enough to contain all the found entries. This value is the number of Exception Log entries that were not copied to the user buffer.
- The Exception Logs entries, which will have the format shown in Table 45

Table 45. Exception Log Entries Returned in User Buffer

Offset	Length	Description	Type
2	10	Date (DD/MM/YYYY)	ASCII
12	8	Time (HH:SS:MM)	ASCII
20	2	ID Node	ASCII
22	1	Action (U = Update; D = Delete; R = Rename)	ASCII
23	25	File Name	ASCII
48	25	Old File Name	ASCII
73	4	Error Code	Integer
77	4	Reconciliation Error Code	Integer
81	2	Location	Integer
83	1	Partial Error Tracking (Y or N)	ASCII
84	1	Key File (Y or N)	ASCII
85	2	Back Level	Integer

**Retrieve File Server Exception Logs:** This function uses the following parameters:

**RET**

	<b>0</b>	No entries in the File Server Exception Logs
	<b>&gt;0</b>	Specifies the number of entries in the File Server Exception Logs
	<b>xxxxxx</b>	An error code indicating why the information cannot be obtained
<b>FUNC-NUM</b>	<b>= 210</b>	
<b>DATA-BUFF</b>	<b>= Buffer to retrieve File Server Exception Logs</b>	
<b>DBUFF-SIZE</b>	<b>= Size of DATABUFF</b>	

Upon a successful return, this will be in the user buffer:

- Two bytes containing an integer value that specifies the number of missing entries
- The Exception Logs entries, which will have the format shown in Table 45 on page 374.

**Check a File in Exception Logs:** This function checks whether there are exception logs for a specific file.

This function uses the following parameters:

**RET**

	<b>0</b>	No entries in the Exception Logs
	<b>1</b>	At least one entry in the Exception Logs
	<b>xxxxxx</b>	An error code indicating why the status cannot be obtained
<b>FUNC-NUM</b>	<b>= 211</b>	
<b>DATA-BUFF</b>	<b>= NULL-terminated character pointer with the file name to search for in the Master or File Server Exception Logs</b>	
<b>DBUFF-SIZE</b>	<b>= Size of DATABUFF</b>	

**Wait for system console keyboard and mouse inactivity:** This function returns when no controller keyboard, terminal Java keyboard, or mouse activity has occurred on the system console for the time specified. Refer to “Programming auto sign off” on page 5 for more information about using Auto Sign Off.

This function uses the following parameters:

**RET**

	<b>-1001</b>	Data buffer size is not valid.
	<b>-1101</b>	The requestor is not a background application.
	<b>-1301</b>	A keyboard and mouse wait is already in progress.
	<b>-1302</b>	Amount of seconds to wait for inactivity ( <b>DATA-BUFF</b> ) is out of the allowed range.
<b>FUNC-NUM</b>	<b>= 20</b>	
<b>DATA_BUFF</b>	<b>= First int (for 16bit) or short (for 32bit) contains the amount of seconds of keyboard and mouse inactivity to wait for, range 0x000F-0x0E10 (15 seconds – 1 hour).</b>	
<b>DBUFF-SIZE_</b>	<b>= 2</b>	

**Wait for system console keyboard or mouse Activity:** This function returns at the first controller keyboard, terminal Java keyboard, or mouse activity on the system console. Refer to “Programming auto sign off” on page 5 for more information about using Auto Sign Off.

This function uses the following parameters:

**RET**

	<b>-1101</b>	The requestor is not a background application.
--	--------------	--

<b>FUNC-NUM</b>	<b>-1301</b>	A keyboard and mouse wait is already in progress.
<b>DATA-BUFF</b>	<b>= 21</b>	
<b>BUFF-SIZE</b>		Unused

**Sign off the system console active user:** This function signs off the system console active user. Refer to “Programming auto sign off” on page 5 for more information about using Auto Sign Off.

This function uses the following parameters:

**RET**

- 1101** The requestor is not a background application.
- 1303** Request only valid when the system is in controller or terminal Java mode.
- 1304** Request not valid when there is not a system console active user.

<b>FUNC-NUM</b>	<b>= 22</b>
<b>DATA-BUFF</b>	Unused
<b>DBUFF-SIZE</b>	Unused

**Disconnect the system console active user:** This function disconnects the system console active user. Refer to “Programming auto sign off” on page 5 for more information about using Auto Sign Off.

This function uses the following parameters:

**RET**

- 1101** The requestor is not a background application.
- 1303** Request only valid when the system is in controller or terminal Java mode.
- 1304** Request not valid when there is not a system console active user.

<b>FUNC-NUM</b>	<b>= 23</b>
<b>DATA-BUFF</b>	Unused
<b>DBUFF-SIZE</b>	Unused

**Start inactivity monitoring:** This function returns when:

- No controller or terminal Java keyboard or mouse activity has occurred on the system console for the time specified, and
- No keyboard activity has occurred on any auxiliary console for the time specified.

Refer to “Programming auto sign off for system and auxiliary consoles” on page 6 for more information about using auto signoff for system and auxiliary consoles.

Supported on enhanced systems only.

This function uses the following parameters:

**funcnum**

= 14

**databuff[0]**

= First int (for 16bit) or short (for 32bit) contains the amount of seconds of keyboard and mouse inactivity to wait for, range 0x000F – 0x0E10 (15 seconds – 1 hour).

**dbuffsize**

= 2

**RET**

= One of the following values:

0x8000 = System console has had no keyboard or mouse activity for the specified amount of time.

0x0001-0x0008 = specified auxiliary console has had no keyboard activity for the specified amount of time.

Both the system console and a single auxiliary console can return at the same time. For example, a return value of 0x8003 would indicate that the system console and auxiliary console 3 have had inactivity for the specified amount of time. All consoles that are not specified in the return value will continue to be monitored for inactivity.

**RET**

One of the following error values:

-1001 = dbuffsize invalid.

-1101 = Requestor not a background application.

-1301 = Keyboard and mouse wait already in progress.

-1302 = Amount of seconds to wait for inactivity (PARM1) out of range.

-1304 = No active user on any consoles (system or auxiliary).

**Stop inactivity monitoring:** This function returns when keyboard and mouse inactivity monitoring for all consoles (system and auxiliary) has been stopped.

Supported on Enhanced systems only.

This function uses the following parameters:

**funcnum**

15

**databuff, dbuffsize**

= Unused

**RET**

= One of the following error values:

-1101 = Requestor not a background application.

-1301 = Keyboard and mouse wait already in progress.

**Signoff specific or all active users:** This function signs off the active users from the requested consoles.

Supported on Enhanced systems only.

This function uses the following parameters:

**funcnum**

16

**databuff[0]**

First int (for 16bit) or short (for 32bit) contains the console to signoff the active user.

0 = System console

1-8 = Auxiliary console 1,2,3,4,5,6,7, or 8

-1 = (0xFFFF) = all consoles (system and auxiliary).

**dbuffsize**

2

**RET**

= One of the following error values:

-1001 = dbuffsize invalid.

-1101 = Requestor not a background application.

-1302 = Invalid console requested.

-1303 = Request not valid for system console when c/t is in terminal mode.

-1304 = No active user on specified consoles.

-1305 = Request not valid when there is no auxiliary console configured for this console.

**Disconnect specific or all active users:** This function signs off the active users from the requested consoles.

Supported on Enhanced systems only.

This function uses the following parameters:

**funcnum**

17

**databuff[0]**

First int (for 16bit) or short (for 32bit) contains the console to signoff the active user.

0 = System console

1-8 = Auxiliary console 1,2,3,4,5,6,7, or 8

-1 = (0xFFFF) = All consoles (system and auxiliary).

**dbuffsize**

2

**RET**

= One of the following error values:

-1001 = dbuffsize invalid.

-1101 = Requestor not a background application.

-1302 = Invalid console requested.

-1303 = Request not valid for system console when c/t is in terminal mode.

-1304 = No active user on specified consoles.

-1305 = Request not valid when there is no auxiliary console configured for this console.

**User Friendly Master/File Server Activation and Deactivation:** This operation provides the following functions:

- Activate Master/File Server
- Deactivate Master/File Server
- Delete Single exception log entry (master or file server)
- Delete All exception log entries ( master or file server )

**16-Bit C Interface:**

```
void ADX_CSERVE(long *ret, int funcnum, char *databuff, int dbuffsize);
```



### 32-Bit C Interface:

```
void ADX_CSERVE(long *ret, short funcnum, char *databuff, short dbuffsize);
```

#### where:

**ret** Return code values of the service requested. An error code is returned if an error occurred.

**funcnum** Function number for the service requested.

**databuff** Buffer for data sent to service requested or data received from service requested.

**dbuffsize** Size (bytes) of data-buff or data for requested service if databuff is not used.

**Delete Master Exception Log Entries:** If the user wants to delete all of the entries or only one from the Master exception log then the **funcnum** parameter value will be 213. According to the above interface definition the function **Delete Master Exception Log Entries** will be defined using the following parameters:

**RET** = 0 Success

**RET** < 0 Error code

**FUNCNUM** = 213

**DATABUFF** = Buffer to specify Master Exception Log entries to delete

**DBUFFSIZE** = Size of DATABUFF

If **DBUFFSIZE** is 0 (zero) then all of the exception log entries will be deleted.

To delete a specific group of exception log entries, **DATABUFF** should contain:

Table 46. Delete Master Exception Log Entries DATABUFF

Offset	Length	Description	Type
0	25	FILE NAME	ASCII
25	1	ACTION ( U - D )	ASCII
26	2	ID NODE	ASCII
28	10	STARTING DATE(DD/MM/YYYY)	ASCII
38	5	STARTING TIME (HH:MM)	ASCII
43	10	ENDING DATE(DD/MM/YYYY)	ASCII
53	5	ENDING TIME (HH:MM)	ASCII

The use of wildcards '\*' is permitted only in the first position (OFFSET) of the field, the function will ignore that field if '\*' is found.

If '\*' is found in a DATE field, the next field of TIME will be ignored.

**Delete File Server Exception Log Entries:** If the user wants to delete all of the entries or only one from the File Server exception log then the **funcnum** parameter value will be 214. The function **Delete File Server Exception Log Entries** will be defined using the following parameters:

**RET** = 0 Success

**RET** < 0 Error code

**FUNCNUM** = 214

**DATABUFF** = Buffer to specify File Server Exception Log entries to delete

**DBUFFSIZE** = Size of DATABUFF

If **DBUFFSIZE** is 0 (zero) then all of the exception log entries will be deleted.

To delete a specific group of exception log entries, **DATABUFF** should contain:

*Table 47. Delete File Server Exception Log Entries DATABUFF*

Offset	Length	Description	Type
0	25	FILE NAME	ASCII
25	1	ACTION ( U - D )	ASCII
26	2	ID NODE	ASCII
28	10	STARTING DATE(DD/MM/YYYY)	ASCII
38	5	STARTING TIME (HH:MM)	ASCII
43	10	ENDING DATE(DD/MM/YYYY)	ASCII
53	5	ENDING TIME (HH:MM)	ASCII

The use of wildcards '\*' is permitted only in the first position (OFFSET) of the field, the function will ignore that field if '\*' is found.

If '\*' is found in a DATE field, the next field of TIME will be ignored.

**Activate Master/File Server:** If the user wants to Activate the Master or the File Server, the **funcnum** parameter value will be 215 for the function **Activate Master/File Server** with the following parameters:

**RET** = 0 Success

**RET** < 0 Error code

**FUNCNUM** = 215

**DATABUFF** = **1** to Activate the Master, **2** to Activate the File Server

**DBUFFSIZE** = Not used

**Deactivate Master/File Server:** If the user wants to Deactivate the Master or the File Server, the **funcnum** parameter value will be 216 for the function **Deactivate Master/File Server** with the following parameters:

**RET** = 0 Success

**RET** < 0 Error code

**FUNCNUM** = 216

**DATABUFF** = **1** to Deactivate the Master, **2** to Deactivate the File Server

**DBUFFSIZE** = Not used

Always verify the **RET** parameter of the ADX\_CSERVE function. This value reports the return code of the service requested. Below are listed all of the possible return codes.

Table 48 on page 381 shows the return codes returned by this function.

*Table 48. Function Return Codes*

Value	Description
0L	Good Return Code
-1001L	Buffer size or buffer address invalid
-1024L	Invalid parameters
-1025L	Non-LAN system
-1026L	Unable to access remote controller
-1027L	Not Master or File Server Acting
-1028L	Special call fails
-1029L	Entries found in Master exception log. Deactivation not allowed.
-1030L	Entries found in File Server exception log. Deactivation not allowed.
-1031L	A Controller is already acting as Master
-1032L	A Controller is already acting as File Server
-1033L	This Controller is not acting as Master
-1034L	This Controller is not acting as File Server
-1035L	Controller not configured for this function.
-1188L	No memory available

See the *4690 OS: Messages Guide* for system errors.

**Eject CD/DVD media:** This function allows an application to programmatically eject CD/DVD media inserted in the controller CD/DVD drive.

**Note:** Using the *Eject* function while writable media is in use may lead to data loss or damage to the media.

In order to prevent data loss, ensure that a *write* operation is not in progress. Issue the **unlockp** command before attempting to eject the media.

New error return codes were added to provide specific results of the Eject command.

This function uses the following parameters:

**FUNC-NUM** = 217

**DATA-BUFF** = Unused. The value is ignored.

**DBUFF-SIZE** = Unused. The value is ignored.

**RET** = One of the following values:

0x805B9700 = A CD/DVD operation was in or is in progress preventing the eject.

0x805B9701 = Drive is locked, you must issue an **unlockp** command prior to attempting the eject.

0x805B9702 = This is a fatal error and it signals that the low CD/DVD device driver was unable to be opened.

0x805B9703 = The unit's door is closing, retry the **eject** command.

0x805B9704 = The CD/DVD Device Driver failed to acknowledge the command.

0x805B9707 = The eject was not performed due to a **format** or **chkdsk** command being in progress.

-1020 = A resource on the P: drive is in use, preventing the **eject** command from completing. Usually this indicates that the working directory on the P: drive is set to something other than the root directory.

0 = Success

## Starting a Background Application

This operation can be called from an application written in C and COBOL to start a background application on the operating system. The ability to set a priority level for the background application is included. This function is valid only for store controller applications. See "ADXSTART" on page 280 for a description of starting as background application from a CBASIC application.

### 16-Bit C Interface:

```
void ADX_CSTARTP(long *pid, char *bacappl, char *parms, int parmlen, char
*initmsg, int msglen, int cpriority);
```

### 32-Bit C Interface:

```
void ADX_CSTARTP(long *pid, char *bacappl, char *parms, short parmlen,
char *initmsg, short msglen, short cpriority);
```

### COBOL Interface:

77	BACAPPL	PIC X(22)	USAGE IS DISPLAY.
77	PARMS	PIC X(46)	USAGE IS DISPLAY.
77	PARMLN	PIC 9(4)	USAGE IS COMP-5.
77	INITMSG	PIC X(46)	USAGE IS DISPLAY.
77	MSGLEN	PIC 9(4)	USAGE IS COMP-5.
77	CPRIORITY	PIC 9(4)	USAGE IS COMP-5.
77	PID	PIC S9(8)	USAGE IS COMP-5.

```
CALL "ADX_CSTARTP" USING PID,
BACAPPL, PARMS, BY VALUE PARMLN, BY REFERENCE INITMSG,
BY VALUE MSGLEN, BY VALUE CPRIORITY.
```

### Parameters:

<b>bacappl</b>	Background application name. The name must be terminated with a NULL or a blank. Maximum length = 22 characters including the ending blank or NULL.
<b>parms</b>	Parameter list to be passed to background application. Note the system automatically passes BACKGRND as the first parameter and appends the parameters in PARMS.
<b>parmlen</b>	Length of parameter list. Maximum length = 46 characters.
<b>initmsg</b>	The initial message to be displayed on the BACKGROUND APPLICATION CONTROL panel when the application is started.
<b>msglen</b>	Length of the initial message. Maximum length = 46 characters.
<b>cpriority</b>	Value for setting priority. Can be from 1 to 9. This value is added to 195 to give the priority 196 to 204. The recommended value is 5 to yield priority 200. A lower value affects other applications that can be running with a different priority.
<b>pid</b>	Return code. It contains the process ID or an error if an error occurred.

Table 49 on page 383 shows unique error codes that you can receive:

*Table 49. Error Codes*

Error Code	Description
-1170	Application name missing or not valid.
-1171	All background application list entries are in use.
-1172	Maximum number of background applications already active.
-1175	Invalid parameter.
-1176	Internal error.
-1177	Specified priority out of range.

For any system errors that are returned, see the *4690 OS: Messages Guide*.

## Logging an Error

This function can be called from a store controller application written in C or COBOL to log an error in the application error log. See “ADXERROR (application event logging)” on page 312 for a description of logging an error from a BASIC application. The function looks for a file of error messages to display along with the unique data. These messages should be in file ADXCISOZF.DAT in the form specified. The application name is automatically included in the log entry.

There is no return code for this function.

### 16-Bit C Interface:

```
void ADX_CERROR(int msggrp, int msgnum, int severity, int event, char *unique,
int ulen);
```

### 32-Bit C Interface:

```
void ADX_CERROR(short msggrp, short msgnum, short severity, short event, char
*unique, short ulen);
```

### COBOL Interface:

77	MSGGRP	PIC 9(4)	USAGE IS COMP-5.
77	MSGNUM	PIC 9(4)	USAGE IS COMP-5.
77	SEVERITY	PIC 9(4)	USAGE IS COMP-5.
77	EVENT	PIC 9(4)	USAGE IS COMP-5.
77	UNIQUE	PIC X(10)	USAGE IS DISPLAY.
77	ULEN	PIC 9(4)	USAGE IS COMP-5.

CALL "ADX\_CERROR" USING BY VALUE MSGGRP,  
 BY VALUE MSGNUM, BY VALUE SEVERITY, BY VALUE EVENT,  
 BY REFERENCE UNIQUE, BY VALUE ULEN.

**Parameters:**

<b>msggrp</b>	Two-byte integer that contains the ASCII equivalent of the message group character used to identify the error messages for a product. This character is unique for each product and can be in the range J to S. Example: MSGGRP = 75 for the letter K.
<b>msgnum</b>	Two-byte integer message number, if any. If the message number is zero, no message is displayed. This number is converted to three printable decimal digits and appended to the message group letter to form the message identifier. This identifier is used to search the Application Message file for a message to display.
<b>severity</b>	Two-byte integer ranging from 1 to 5 that is used to indicate the importance of each message. This number is a message level indicator with the most important messages as severity = 1. An operator can request messages to be displayed. The messages with severity level less than or equal to the level requested is displayed.
<b>event</b>	Two-byte integer that can be set by the application to indicate a specific situation or problem. This must be in the range 64 to 79 for the controller. The system uses the log data to generate Network Problem Determination Alert messages.
<b>unique</b>	Short string of characters to be included in message. Maximum length is 10 bytes. If the message is longer than 10 bytes, only the first 10 bytes is used. If the message is less than 10 bytes, blanks are added.
<b>ulen</b>	Number of characters in unique message including imbedded blanks.

## Miscellaneous Services

The Miscellaneous Services functions provide a variety of functions to end a program, exit a program, get additional storage, and so on.

### Ending a Program

This operation removes a process from the system. Any outstanding events for the process are canceled, opened files are closed, and memory is released. A process can only be ended by another process with the same user and group or by a superuser.

**16-Bit C Interface:**

```
void ADX_CABORT(long *ret, long pid);
```

**32-Bit C Interface:**

```
void ADX_CABORT(long *ret, long pid);
```

**COBOL Interface:**

77	PID	PIC S9(8)	USAGE IS COMP-5.
77	RET	PIC S9(8)	USAGE IS COMP-5.

CALL "ADX\_CABORT\_PROS" USING RET, BY VALUE PID.

**Parameters:**

<b>pid</b>	Process ID of target process to end.
<b>ret</b>	Return code. An error code is returned if an error occurred.

See the *4690 OS: Messages Guide* for information on the error codes returned by this function.

## Exiting a Program

This operation terminates the program, returns control to the operating system, and passes back the completion status value to the calling program. Any outstanding events are canceled and open files closed.

The low order 16-bit word of the status can be used by an application to indicate status when the application was exited. Bits 0–14 of this application status word are available to the application. By convention, a 0 value is used to indicate success while positive values indicate some form of partial completion. Bit 15 is set by the operating system (making the application status word negative) when the attempt to create the process resulted in an error or the process was abnormally terminated. This error can be used if the program is started from a batch file.

There is no return code for this function.

### **16-Bit C Interface:**

```
void ADX_CEXIT(long estat);
```

### **32-Bit C Interface:**

```
void ADX_CEXIT(long estat);
```

### **COBOL Interface:**

```
77  ESTAT          PIC S9(8)          USAGE IS COMP-5.  
  
CALL "ADX_CEXIT_PROS" USING  BY VALUE ESTAT.
```

### **Parameters:**

**estat** A 32-bit value. The low order 16-bit word is the completion status. The high order 16-bit word is reserved.

Bits 0–14 can be used to indicate completion status.

Bits 15–31 are reserved and must = 0.

## Getting Additional Storage

This operation either adds contiguous memory to the end of an existing heap or allocates a new heap. Use the option field to select one or the other and the Memory Parameter Block to specify the minimum and maximum memory requirements. Set the Memory Parameter Block's start parameter to the base address of the existing heap for option 0 or to zero for option 1.

**Note:** Processes are not automatically given an initial heap allocation. Consequently, option 1 must be called the first time that heap space is needed.

When option 0 is selected, the designated heap is extended contiguously. The Memory Parameter Block start address (start) and minimum allocation (min) are modified to reflect the new starting address and actual allocation, respectively. The original heap's base address and contents remain unchanged.

When option 1 is selected, the new heap might or might not be contiguous with any previously allocated heap. ADX\_CMEM\_GET modifies the Memory Parameter Block's start and min values to indicate the new heap's base address and actual allocation. The new heap can be allocated such that an existing heap is no longer expandable.

### **16-Bit C Interface:**

```
void ADX_CMEM_GET(long *ret, int option, long mpbptr[3])
```

**Note:** A COBOL interface is not available for this operation.

### **Parameters:**

**option**

0 = Expand existing heap  
 1 = Allocate a new heap

**mpbptr**

Address of Memory Parameter Block.

**ret**

Return code. An error code is returned if an error occurred.

**32-Bit C Interface:** These functions are not supported in the 32-bit interface. Memory should be allocated/freed using the C run-time memory allocation functions.

**Memory Parameter Block:**

Byte

0

start

4

min

8

max

**start**

For option equal 0, set the base address of the heap segment to be expanded in this field. The base address of the added memory portion is put here when the allocation has completed. For option equal 1, set this field to zero and the base address of the new heap is put here.

**min**

Specify the minimum number of bytes required. The actual number allocated is returned.

**max**

Specify the maximum number of bytes required. This value is not changed.

See the *4690 OS: Messages Guide* for information on the error codes returned by this function.

**Freeing Storage**

This operation releases the memory in a heap from the address specified to the end of the heap.

**16-Bit C Interface:**

```
void ADX_CMEM_FREE(long *ret, void *mstart)
```

**Note:** A COBOL interface is not available for this operation.

**Parameters:****mstart**

Beginning address of heap to free.

**ret**

Return code. An error code is returned if an error occurred.

See the *4690 OS: Messages Guide* for information on the error codes returned by this function.

**32-Bit C Interface:** These functions are not supported in the 32-bit interface. Memory should be allocated/freed using the C run-time memory allocation functions.

**Suspended Processing for Indicated Duration**

This operation delays the calling process until the specified time or until the specified period of time expires.



If absolute time is specified and the current time of day is beyond it, the process delays until the specified time on the next day.

#### **16-Bit C Interface:**

```
void ADX_CTIMER_SET(long *ret, unsigned int flags, long stime);
```

#### **32-Bit C Interface:**

```
void ADX_CTIMER_SET(long *ret, unsigned short flags, long stime);
```

#### **COBOL Interface:**

```
77  FLAGS          PIC 9(4)          USAGE IS COMP-5.  
77  STIME          PIC 9(8)          USAGE IS COMP-5.  
77  RET            PIC S9(8)         USAGE IS COMP-5.
```

```
CALL "ADX_CTIMER_SET" USING RET,  
BY VALUE FLAGS, BY VALUE STIME.
```

#### **Parameters:**

##### **flags**

1 — absolute  
0 — relative

##### **stime**

If FLAGS = 1 (absolute), stime indicates the number of milliseconds to delay after midnight.  
If FLAGS = 0 (relative), number of milliseconds to delay.

**ret** Return code. An error code is returned if error occurred.

See the *4690 OS: Messages Guide* for information about the error codes returned by this function.

## **Waiting for Event Completion**

An application can initiate a WAIT. This operation waits for data to be available in a pipe or for a change of status from the host. Several WAITs can be initiated in one call. The first WAIT to be satisfied indicated in the return code. The WAIT also terminates if the specified time to wait elapses before any event completes.

If more than one WAIT is satisfied at the same time, the one that occurs first in the parameter list is indicated in the return code. Thus, the order of the parameters in the list can be significant. If no time limit is specified, the operating system waits indefinitely until at least one of the waits is satisfied.

For pipes, the wait for data to be available in a pipe is satisfied when at least one byte is in the pipe. For change in status from the host, the application must have requested the status before calling for a wait because the change is considered as change since the status was last requested.

The maximum number of waits that can be requested at a time is 30. However, the OS also uses waits, so this maximum cannot be available. If more waits are requested than are available, the application ends and causes a dump if the dump is enabled.

#### **16-Bit C Interface:**

```
struct waitblk  
{  
    long fnum;  
    char wtype;  
} wblk[n]; /*n == number of waits required by the application. */  
void ADX_CWAIT(long *ret, long wtime, int wcount, struct waitblk wblk[]);
```

#### **32-Bit C Interface:**

```
void ADX_CWAIT(long *ret, long wtime, unsigned short wcount, WAITBLK *wblk);
```

### COBOL Interface:

```
01 WAITBLK OCCURS n TIMES INDEXED BY NWAITS.  
*   Where n is the maximum number of waits that the  
*   application requires.  
    02 FNUM          PIC S9(8)          USAGE IS COMP-5.  
    02 WTYPE         PIC 9(2)           USAGE IS COMP-5.  
77 RET              PIC S9(8)          USAGE IS COMP-5.  
77 WTIME            PIC S9(8)          USAGE IS COMP-5.  
77 WCOUNT          PIC 9(4)           USAGE IS COMP-5.  
77 WBLK             PIC 9(4)           USAGE IS POINTER.  
  
SET WBLK TO ADDRESS OF WAITBLK (1).  
CALL "ADX_CWAIT" USING RET,  
BY VALUE WTIME, BY VALUE WCOUNT, BY VALUE WBLK.
```

### Parameters:

**wtime** Time to wait, in milliseconds. For example, to wait 1 second set `wtime = 1000`.

**Note:** 0000 = infinite.

### wcount

Count of *fnums* entered in *wblk*. This should be the actual count of wait events to initiate and not the maximum. For most applications this count is less than 10 and usually 1 or 2. Maximum count = 30.

**fnum** For pipes, *fnum* is the file number or pipe identifier returned when the pipe was created. For host communications, *fnum* is the link number or session number returned when the link or session was opened.

**wtype** Type of event for wait. (This is a one-byte integer, not an ASCII character.)  
1 = pipe  
2 = host

Any other value causes an error to be returned.

### wblock

A pointer to an array of *waitblk* data structures that contain the data to wait on. This array should be packed on 1-byte boundaries. For example, each element of the array should be 5 bytes in length and there should be no padding between array elements. (In 32-bit mode, including CAPIC.H and using an array of type *WAITBLK* provides the proper byte packing.)

**ret** Return code. If *ret* is positive an event completed and *ret* is the index number of the completed event. For example, if *wcount* = 2 when *ADX\_CWAIT* is called, *ret* = 1 indicates the first event completed, *ret* = 2 indicates the second event completed. If *ret* = 0, then the specified time expired before an event completed. If *ret* is negative, then an error occurred.

Table 50 on page 388 shows the error codes unique to this function.

Table 50. Unique Error Codes

Error Code	Description
-1000	<i>wtime</i> < 0
-1001	<i>wcount</i> <= 0 or <i>wcount</i> > 30
-1002	<i>wtype</i> value not valid

See the *4690 OS: Messages Guide* for any system error that is returned.

### Converting HEX to ASCII

This function converts four hexadecimal bytes to eight bytes of ASCII characters. This function can be used to convert the 4690 system error codes from hexadecimal characters to printable ASCII characters.

### 16-Bit C Interface:

```
void ADX_CPRTHEX(char *anum, long hnum);
```

### 32-Bit C Interface:

```
void ADX_CPRTHEX(char *anum, long hnum);
```

### COBOL Interface:

```
77 ANUM          PIC X(8)          USAGE IS DISPLAY.  
77 HNUM          PIC S9(8)         USAGE IS COMP-5.  
  
CALL "ADX_CPRTHEX" USING ANUM, BY VALUE HNUM.
```

### Parameters:

**anum** Buffer to receive ASCII characters. It must be eight bytes.

**hnum** Hexadecimal number to convert. It must be four bytes.

---

## Guidelines and Restrictions for 16-Bit Assembly Language Applications

- All applications must be relocatable.
- Do not use instructions that are restricted to 80286 or 80386 privileged mode. These instructions are:
  - IN/INS and OUT/OUTS
  - CLI and STI
  - HLT
- Do not address any absolute memory location such as:
  - Interrupt vector table (0000-03FF)
  - ROM communication area (400-5FF)
  - Display screen buffers (B0000-B3FFF and B8000-BBFFF)
  - Physical addresses for memory mapped I/O adapters
  - BIOS routines
- Do not attempt to address any area of memory that has not been assigned to the application either at load time or by a dynamic memory allocation.
- Do not load anything but a valid address that has been provided by the system into a segment register.
- Do not perform address arithmetic in a segment register.
- Do not depend on the initial value stored in any segment register for purposes other than addressing. Do not assume the value of one segment by examining the value of another segment.
- Do not attempt to execute code in a data segment and do not attempt to write data in a code segment.
- Language subroutines addressed by the interrupt vector table must be reentrant to allow for multiple accesses from applications executing at the same time.
- Non-reentrant subroutines must be addressed by call mechanisms linked with the application that is using the subroutines to ensure a unique copy of the subroutines for each application.

Following are the currently known differences between the 80286 or 80386 and the 8088 or 8086 instructions. You can reference the currently available Intel™ documents for the details of the following items and for any additional differences.

Some of these functional differences can be handled by the operating system; however, all of the known differences are listed here.

The following differences are handled by the operating system:

- The interrupts that only occur for instructions that are new in the 80286 or 80386 or for protection exceptions. The interrupts are for:

Interrupt Number	Description
5	BOUND instruction fault
6	Undefined opcode attempted
7	Processor extension not available

<b>8</b>	Double fault
<b>9</b>	Processor extension segment overrun
<b>10</b>	Invalid Task State Segment (TSS)
<b>11</b>	Segment not present
<b>12</b>	Stack exception
<b>13</b>	General protection exception
<b>16</b>	Math fault

- Divide exceptions point at the DIV instruction.
- Do not single step external interrupt handlers.
- Do not rely on NMI interrupting NMI handlers.

The following differences can impact an application:

- Most instructions take fewer clock cycles in the 80286 or 80386 than in the 8088 or 8086.
- PUSH SP in the 80286 puts the value of SP before the instruction was executed onto the stack. POP SP works accordingly.
- The 80286 masks all shift and rotate counts to the low 5 bits (maximum of 31 bits).
- Do not duplicate prefixes. The 80286 sets an instruction length limit to 10 bytes.
- Do not rely on IDIV exceptions for quotients of 80h or 8000h.
- Instructions or data items might not wrap around a segment.
- Do not attempt to change the sense of any reserved or unused bits in the flag word by IRET.

---

## Chapter 18. Using IBM DOS Applications

**Note:** IBM DOS Applications work *only* on the 80286 processor.

The operating system includes an interface that emulates the environment of the IBM Personal Computer Disk Operating System (IBM DOS) Version 2.1. This chapter gives you the information you need to run or write applications for that environment. The chapter contains information on software interrupts, Personal Computer Basic Input/Output System (PC BIOS) calls, and guidelines for applications written to run under the IBM DOS environment.

A IBM DOS application should run under the 4690 DOS emulation provided:

- It meets the guidelines for IBM DOS applications as described
- It uses only the hardware listed.

To help you determine if an application is suitable for the 4690 DOS emulator, four optional debug reports are available. The four reports are activated using the DEFINE command and setting the EOPTIONS parameter.

---

### Starting Applications in the DOS Environment

The 4690 DOS emulator executes a DOS application in the protected mode. In this mode an application is restricted from the execution of some machine instructions and is restricted from accessing any memory that is not initially assigned to it.

To run in the operating system, DOS applications must conform to the following guidelines.

The program **must**:

- Use DOS function calls to perform all system functions and data I/O.
- Use the DOS system call to obtain additional memory.
- Limit direct BIOS calls (interrupt requests to the BIOS) to the following functions:
  - 10H 13H
  - 11H 16H
  - 12H 17H

The program **must not**:

- Use instructions that are restricted to the privileged mode. These instructions include:
  - IN, INS, OUT, and OUTS except to the CRT controller
  - CLI and STI (the emulator ignores these instructions)
  - LOCK (causes program termination)
  - All unique protection control instructions (causes program termination)
- Jump to or call BIOS routines directly.
- Address DOS flags, data buffers, tables, and work areas.
- Address any memory that has not been assigned to the application at load time.
- Set the video display mode any way other than with BIOS int 10H, subfunction 0 or by accessing the CRT controller with an OUT instruction.
- Define the DOS reserved areas of the PSP and FCB.
- Depend on the environment string.

Generally, you should not access absolute addresses in memory directly. The only exceptions are the following virtual hexadecimal addresses:

- Screen region buffers B0000–B3FFF (monochrome) and B8000–BBFFF (color)
- Zero page 0–5FF; however, do not depend on the values in 400–5FF.

For better program performance, avoid instructions that load segment registers (SR). These include:

```
LDS
LES
POP sr
MOV sr,...
CALLF
RETF
JMPF
```

In addition, small memory model programs perform better than programs written in other memory models.

---

## DOS Program Memory Allocation

DOS is a single-tasking operating system. Because only one task is intended to run at a time, DOS allows a single program to occupy all of user memory. There is no need to share memory because there is never another program with which to share it.

Because the operating system is a multitasking system in which several programs run at the same time, memory must be carefully allocated. Use the ADDMEM parameter to allocate memory, if needed.

The assumption made by the emulator is for a 128 KB (DOS) machine. If the DOS program is designed to work in a 128 KB or less memory, no memory adjustments are required. If the program requires more than 128 KB, you must define the ADDMEM parameter with the correct machine size.

---

## Allocating Memory Using ADDMEM

The ADDMEM parameter of the DEFINE command lets you control the size of the emulated DOS machine. The system applies the memory allocation on a process family (FID) basis. An ADDMEM memory allocation applies only to programs loaded under the window on which the DEFINE command was invoked. Regular users of DOS applications should include a DEFINE ADDMEM command in a batch (BAT) file used to invoke the application.

The DEFINE command used to define the DOS machine has the following form:

```
DEFINE addmem=n
```

Where *n* specifies the amount of memory in kilobytes.

If the program fails to load because of a lack of memory, a message indicating the amount of memory required is displayed. The message looks like:

```
Code: zzzzzzzz - Not enough memory.  Need xxx KB.
Requested machine size was: yyy KB.
```

```
Run has been aborted.
```

Where xxx and yyy are numbers appropriate to the situation. An ADDMEM value of xxx should permit the program to load. Code zzzzzzzz is explained in “Emulator Messages” on page 396.

---

## Optional Emulator Reports

It is difficult to determine if a program is appropriate for 4690 DOS emulation. The optional reports slow the execution of a DOS program but they can provide the data needed to identify unsupported function or explain unusual results.

To activate the reports, you must define the EOPTIONS parameter with the appropriate report selected; otherwise no reports are created.

---

## Selecting Reports Using EOPTIONS

The EOPTIONS parameter of the DEFINE command enables you to select one or more reports that are written during the execution of all subsequent DOS applications. For each execution of a DOS application, the previous reports are erased.

All reports are added to a file called E\_DATA in the root directory of the C: drive. You must rename, copy, or move this file if you want to keep the results of the DOS application execution.

The DEFINE command used to select all reports has the following form:

```
DEFINE eoptions=abcd
```

Where *abcd* must be “yyyy” to activate all (4) available reports. An “n” in any position prevents that report.

### Report “a”—Errors

Report “a” records all errors encountered by the emulator. An example of this output is:

```
INT21 Subfunction 49h illegal.  
Execution has been aborted.
```

### Report “b”—Startup Data

Report “b” produces a display at startup time showing the initial conditions for the emulated program. This display is also recorded in the report file and could look like:

```
Emulation Options are yyyy  
  
DOS   E M U L A T I O N
```

Environment:

```
Program name:  h0:/edit.exe  
Input data:    ''  
Machine size:  128 KB  
Display Type:  Color  
286 installed?: Yes  
# of floppies: 1  
# of hardfiles: 1
```

```
Do you wish to continue? (y/n):
```

### Report “c”—Interrupt Trace

Report “c” produces a report tracing all interrupts and recording the registers before and after the interrupt. The trace report could look like:

CS	IP	Service	AX	BX	CX	DX	DS	SI	ES	DI	SS	SP	BP	FL
1C0E:002B	INT 11h		16490000	00001000	16490000	16490000	10100289	0000	0293					
1C0E:002B	INT 11h		00230000	00001000	16490000	16490000	10100289	0000	0293					
1C0E:0032	INT 21h		19230000	00001000	16490000	16490000	10100289	0000	0293					
1C0E:0032	INT 21h		19020000	00001000	16490000	16490000	10100289	0000	0292					

### Report “d”—OUT, OUTS, IN, INS Trace

Report “d” produces a report tracing all IN, INS, OUT, and OUTS and record the registers before the operation. The trace report could look like:

CS	IP	Service	AX	BX	CX	DX	DS	SI	ES	DI	SS	SP	BP	FL
103A:5172	OUT Eeh		00070304	07070305	16491020	08000000	10100271	0000	0206					
103A:5179	OUT Eeh		00000305	07070304	16491020	08000000	10100271	0000	0202					

---

## DOS BIOS Calls and Software Interrupts

Following is a list of the DOS BIOS software interrupts supported.

**Note:** You cannot mix DOS calls with operating system calls in the same program.

### BIOS Calls

**Note:** Only BIOS calls and subfunctions documented in this section are supported. All others are unsupported.

#### Int 10H, Subfunction n:

- 00H:** Supported. Can also set mode by writing to the CRT controller.
- 01H:** Supported. Sets no cursor if no cursor or a bad cursor value is specified else sets blinking line cursor.
- 02H:** Supported. The video page field is ignored.
- 03H:** Supported. The video page field is ignored.
- 06H:** Supported.
- 07H:** Supported.
- 08H:** Supported. The display page is ignored.
- 09H:** Supported. The display page is ignored.
- 0AH:** Supported. The display page is ignored.
- 0FH:** Supported. The active display page is always zero.

#### Int 11H:

Supported. It returns 287 presence and initial video mode plus the following static values:

- IPL present
- One disk drive
- High byte always 0

#### Int 12H:

Supported. Returns memory allocated to the current process.

#### Int 13H, Subfunction n:

- 00H:** Supported.
- 01H:** Supported.
- 02H:** Supported.
- 03H:** Supported with restrictions to maintain system integrity.
- 04H:** Supported.

#### Int 16H, Subfunction n:

- 00H:** Supported.
- 01H:** Supported.
- 02H:** Returns zero.

#### Int 17H:

Supported.



## Software Interrupts

### Int 20H:

Supported.

### Int 22H:

Supported.

### Int 23H:

Supported.

### Int 24H:

Supported.

### Int 25H:

Supported.

### Int 26H:

Supported with restrictions to maintain system integrity.

---

## DOS Function Calls

DOS function calls are supported as defined in the following (ascending) ordered list. Gaps in the ascending order imply that the missing function is ***not*** supported.

### Int 21H, Subfunction n:

#### 00H-0CH:

Supported.

#### 0EH-15H:

Supported.

**16H:** Supported without the ability to create read-only files.

**17H:** Supported.

**18H:** Returns zero.

#### 19H-1CH:

Supported.

#### 1DH-1EH:

Returns zero.

**20H:** Returns zero.

**21H:** Supported.

#### 23H-2AH:

Supported.

#### 2CH-2DH:

Supported.

#### 2FH-30H:

Supported.

**33H:** Set has no effect; Get always returns TRUE.

#### 35H-36H:

Supported.

**37H:** Set has no effect; Get returns constant values.

#### 39H-3AH:

Supported.

**3BH:** Supported, defines default: at the process level.

**3CH:** Supported, cannot create read-only files.

**3DH-43H:**  
Supported.

**44H:** Subfunctions 0, 1, 6, 7 are supported.

**45H-47H:**  
Supported.

**48H:** Effective if function 7AH is first called to shrink memory allocation by the size that is wanted.

**49H-4AH:**  
Supported.

**4CH:** Supported.

**4DH:** Returns zero. Processes continue asynchronously.

**4EH-51H:**  
Supported.

**54H:** Returns zero.

**55H-57H:**  
Supported.

---

## Emulator Messages

Potential emulator messages follow:

### Informational

Execution has been ended.

**Warning:** "Greater than 640 KB" request has been reduced to 640 KB.

### Warning

INTyy Subfunction xxh ignored

INTyy ignored

IN instruction to port xxh ignored

OUT instruction to port xxh ignored

OUT instruction. DX reg xxh ignored

OUT instruction. Op code xxh ignored

**Fatal** INTyy Subfunction xxh illegal.

INTyy illegal

Code: zzzzzzzz — Program not found. Try again.

Code: zzzzzzzz — Not enough memory. Need xx KB. Requested machine size was: xx KB.

Code: zzzzzzzz — Bad environment.

Code: zzzzzzzz — User requested termination.

Code: zzzzzzzz — Bad (EXE or COM) file format.

**Note:** zzzzzzzz — if negative is described in the *4690 OS: Messages Guide*. Positive numbers indicate the internal (to the emulator) error number. The code should be reported if an emulation error is suspected.

---

## Chapter 19. Designing Terminal Applications With C Language

This chapter contains coding guidelines and restrictions when you are writing a terminal application in C language to run under the operating system. The C Programming Interface (CPI) provides essentially the same functionality inherent in the 4680 BASIC language set.

**Note:** A CBASIC or 16-bit C application cannot access long file name support either on the controller or from the terminal.

---

### Operating System Interfaces for C

The CPI interfaces closely match those offered by BASIC and, where possible, accept similar parameters. The general functions provided by the CPI include:

- | • An interface to the 469x, SurePOS 300/700 or TCxWave 6140 Series terminal I/O.
- | • An interface to the 469x, SurePOS 300/700 or TCxWave 6140 Series terminal file, pipe, and pipe routing services.
- | • An interface to the 469x, SurePOS 300/700 or TCxWave 6140 Series terminal application services.
- | • An interface to the 469x, SurePOS 300/700 or TCxWave 6140 Series terminal extended memory management routines.

### 16-Bit CPI Libraries

The 16-bit CPI supports the 16-bit programming model. It is implemented as two libraries: one for medium memory model programs and one for large memory model programs.

Medium memory model programs contain one or more code segments and a single data segment. Application and run-time static data, heap, and stack all reside in this data segment.

Large memory model programs contain one or more code segments and multiple data segments. The stack is a single segment. The heap occupies one or more data segments as required. Big model programs place all static data into a single segment, whereas large model programs can assign static data to multiple segments.

If the application is such that it fits the medium memory model, it occupies less space in memory and can have improved performance over the equivalent program compiled and linked in a larger memory model.

The medium model library is named CAPITMED.L86; the large model library is named CAPITLRG.L86. These are non-shared libraries. Any application that includes the CPI must include the relevant library during the link edit process so that it forms a part of the final executable module.

### 32-Bit CPI Libraries

The 32-bit version of the CPI libraries supports a flat-mode 32-bit programming model. All pointers are 32-bit values and the C “int” types are all 32 bits in length. Applications using these functions must be linked with the CAPIT.LIB import library. See Chapter 22, “Creating 32-Bit Programs Using VisualAge C/C++,” on page 665 for more information on creating 32-bit applications.

The 16-bit CPI library was not designed to be thread-safe; however, changes have been made to the 32-bit CPI library to make it safer in multi-threaded applications. Therefore, `adx_errn` and `adx_errf` are not defined as global variables, but are defined as macros. The macros call functions that return the addresses of a per-thread storage location and then dereference the address. From an application, `adx_errn` and `adx_errf` can be treated like global variables. However, the values in them are unique per thread. When a CPI function is called in one thread, it does not affect the `adx_errn` values seen by other threads. In order for an application to use these variables, the file `CAPITHDR.H` must be included.

**Note:** The CPI memory routines are not supported in the 32-bit interface. Use the standard C run-time memory allocation routines instead.

## Error Handling

All CPI functions return a negative value (-1) if an error occurs. Two global variables are defined by the CPI and exported so that they can be interrogated by the application when an error occurs.

These variables are:

**adx\_errn**

A 4-byte error code.

**adx\_errf**

The two-byte session number (if there is one) for the failing operation, or the file number for Extended Memory Management Routines.

The majority of the error codes placed in *adx\_errn* are those returned on processing of a Supervisor call to the Operating System. These codes are negative. Additional error codes can be generated by various components of the CPI. These error codes are positive. These codes are listed in “CPI Error Codes” on page 450.

**Note:** One exception to this rule is the behavior of the Extended Memory Management routines described in “CPI Extended Memory Management for the 469x POS Terminal” on page 445. These routines directly return a negative error code and can also place a code in *adx\_errn*.

## Asynchronous Error Interrupt Handling

The following devices support asynchronous operation:

- Printer stations
- Coin Dispenser
- Serial I/O

The CPI provides a function that is the equivalent of BASIC’s ON ASYNC ERROR CALL statement. This allows an application to define a function that receives control whenever an asynchronous error associated with one of the above devices occurs.

## Using the Heap Manager

The 16-bit CPI contains heap management routines for dynamic allocation and deallocation of heap memory. In the terminal environment, these CPI routines should be used in place of the standard C library functions *malloc*, *calloc*, *realloc*, and *free*. The CPI contains four functions for heap management:

- Allocate memory from the heap
- Free previously allocated heap memory
- Query the total currently available free heap space
- Query the largest contiguous area of free heap space.

These routines are also used by the API itself whenever temporary memory is required.

**Note:** The CPI memory routines are not supported in the 32-bit interface. Use the standard C run-time memory allocation routines instead.

## 16-Bit Restrictions

Standard C library input and output functions such as *fopen*, *printf*, should not be called from 4690 terminal applications that link with these libraries. Similarly, standard memory management functions such as *malloc* and *free* should not be called.

**Note:** The restrictions above do not apply to the 32-bit interface. See Chapter 22, “Creating 32-Bit Programs Using VisualAge C/C++,” on page 665 for more information about restrictions when creating programs to run on terminals.

## Header File

In both 16-bit and 32-bit modes, the header file, `CAPITHDR.H` should be included with your application source modules. This file provides CPI prototypes that aids in detecting improper use of CPI calls within your application.

## Compiling and Linking

**Attention:** This section is valid for the 16-bit interface only.

C applications can be compiled using Metaware High C 1.6 or later. Your modules should be linked using the operating system Linker Utility (LINK86). See Chapter 9, “Using the Linker Utility and the POSTLINK Utility,” on page 207 for more information on using the Linker Utility.

In medium model applications, where dynamic data shares a single data segment with the stack and static data area, you can maximize the segment size with LINK86 option `[DATA[MAX[FFF]]]`. This creates the largest possible heap, but, because it is slightly less than 64 KB, prevents wrapping the stack. The medium model stack is fixed at 2048 bytes and occupies the bottom of the data segment.

The large model stack is a single segment. You can either specify a preferred stack size with the LINK86 option `[STACK[MAX[nnn]]]`, or use the linker default, which varies with the version of LINK86. Check the LINK86 output messages for actual stack size allocated. In order to leave more space for dynamic data (the heap), the large model stack size should be no larger than is necessary for the application to run correctly.

The order of linking libraries to support the 469x, SurePOS or TCxWave family CPI is important. To prevent linker errors, the 4690 FLEXOS library should always be specified after the CPI library in the link. For example:

```
link86 test, capitmed.186[s], hcme.186[s]
```

or

```
link86 test, capitlrg.186[s], hc1e.186[s]
```

---

## File Services

### File Types

In the system the various types of supported files are categorized as follows:

1. Disk Files
  - POS
    - Keyed
    - Non-keyed
      - Random
      - Direct
      - Sequential
  - Non-POS
2. In-memory files
  - Pipes
    - Pipe Routing Services (PRS) pipes

- Non-PRS pipes
- RAM disk

POS files are distinguished from non-POS files by their distribution property. Only POS files can be distributed across the store system LAN to other controllers.

Keyed files are POS files that are formatted in a unique manner that enables records to be accessed using a key related to the data in the file.

RAM disk files can be accessed in the same manner as normal disk files by using the appropriate drive identifier.

Pipes are in-memory files used to communicate between applications. Communications between applications in the same terminal use pipes created and managed by the functions described in “File and Pipe Services” on page 403. To exchange data between applications in different terminals or between controller and terminal, you must use pipes created through PRS.

Some CPI functions are specific to a particular type of file. Other functions can be used for several types of files.

## Access Modes

Access modes determine if a file can be shared. If the file is shared, the following modes are selected when the file is opened:

- Exclusive access by the calling process
- Allows reads by other processes
- Allow reads and writes by other processes

Exclusive access to a file in one process prevents any other process from sharing the file. Exclusive access to a file is denied if another process has the file open.

An attempt to open a file under any of the following three conditions is denied and an error returned:

- Open a file that is already open in exclusive mode
- Open a file in exclusive mode that is already open
- Open a file in write mode that is already open in read mode

## File Security

File security is defined by the file security word (FSW) in effect when a pipe or file is created. Bit settings in the FSW determine read, write, and delete privileges for the file owner, group and world. The default FSW permits unlimited privileges. See “Protecting files” on page 33 for a discussion of file security.

## File Pointers

The operating system supports both sequential and random access to disk files by using the file pointer. The pointer position is initialized to zero when the file is opened or created, and is incremented by the number of bytes read or written on each subsequent file access.

An offset into the file is specified for each read or write of a non-keyed file. This offset can be relative to the pointer position, the beginning of the file, or the end of the file. For sequential access, this offset is set to zero relative to the current pointer position.

Sharing of file pointers between processes is not supported.

**Note:** In multi-threaded 32-bit applications, file handles and pointers are shared between all threads in a process.

## Common File Functions

This section explains functions that are common to most file types.

### Setting File Security

Setting file security determines the owner/group/world privileges for all pipes and files created by the application.

The default file security word (FSW) permits unlimited read/write/delete access to a new pipe or file. The application can call *adx\_tumask()* to define a new FSW, which remains in effect for all disk files and pipes created by the application until it issues another call to *adx\_tumask()*.

#### 16-Bit Syntax:

```
unsigned int adx_tumask(unsigned int newmask);
```

#### 32-Bit Syntax:

```
unsigned short adx_tumask(unsigned short newmask);
```

#### Parameters:

##### **newmask**

The new FSW. The following constants can be OR'd to create *newmask*:

```
#define READ_W    0x0800    // world
#define WRITE_W   0x0400
#define DELETE_W  0x0100
#define READ_G    0x0080    // group
#define WRITE_G   0x0040
#define DELETE_G  0x0010
#define READ_O    0x0008    // owner
#define WRITE_O   0x0004
#define DELETE_O  0x0001
```

The function return value is the previous FSW. It can be used as the argument on a subsequent call to *adx\_tumask* to restore the previous FSW.

### Deleting a File

There are two file delete functions: *adx\_tdelete\_file* and *adx\_tdelete\_fnum*. *adx\_tdelete\_file* deletes a file that is not open or that is on terminal RAM disk. It passes the file name to the operating system. If the file is open anywhere in the system, the file is not deleted until it has been closed.

If the application has the file open and the file is not on terminal RAM disk, it should call *adx\_tdelete\_fnum*, which passes the file number to the operating system. This function fails if the file is open anywhere else in the system. This function is useful when the application opens a file exclusively and then deletes it in order to prevent another program from opening the file in the interval between closing and deleting it.

A PRS pipe is a temporary pipe and is deleted when all the processes using the pipe for communication are terminated.

If the operation is successful, the function returns zero; if not, the function returns -1 and the error code is placed in global variable *adx\_errn*.

**Note:** Use this operation also to delete a non-keyed file or pipe.

#### 16-Bit Syntax:

```
int adx_tdelete_fnum (int session);
```

**32-Bit Syntax:**

```
short adx_tdelete_fnum (short session);
```

**Parameters:****session**

Session number of the open file.

**16-Bit Syntax:**

```
int adx_tdelete_file (char _far *filename);
```

**32-Bit Syntax:**

```
short adx_tdelete_file (char *filename);
```

**Parameters:****filename**

Valid file name.

**Renaming a File**

This function renames a disk file. Read-only files or files that are currently open cannot be renamed. If the new name specifies a different directory, the file is moved provided the new directory is on the same drive. Pipes cannot be renamed.

If the operation is successful, the function returns zero; if not, it returns -1 and the error code is placed in global variable *adx\_errn*.

**Note:** Use this operation also to rename a non-keyed file.

**16-Bit Syntax:**

```
int adx_trename_file(char _far *oldname, char _far *newname);
```

**32-Bit Syntax:**

```
short adx_trename_file(char *oldname, char *newname);
```

**Parameters:****oldname**

Current file name or a previously defined logical name. It should include the path specification when necessary. The maximum length of a file name in the controller is 25 bytes, including the NULL terminator.

**newname**

New file name or a previously defined logical name. It should include the path specification when necessary. The maximum length of a file name in the controller is 25 bytes including, the NULL terminator.

**Getting or Setting File Pointers**

This function either interrogates the current file pointer position or sets a new position. To interrogate the current position, set *flags*, bits 8 and 9 to 01 (relative to file pointer) and set *offset* to 0. Any other combination of values for bits 8-9 and the offset changes the pointer position.

The offset specified in *offset* can be positive or negative. However, if the resulting pointer position is less than zero or greater than the current file size, an error is returned. If the file consists of multibyte records, the offset must fall on a record boundary in order for a subsequent read or write to succeed.

This operation is not valid for pipes or keyed files.



If the operation is successful, the file pointer position is returned. A zero return value indicates that the file pointer is at the beginning of the file.

A negative return value (-1) indicates that an error has occurred. The error code is placed in global variable *adx\_errn*, and the file number is placed in global variable *adx\_errf*.

**16-Bit Syntax:**

```
long adx_tfseek (int session, unsigned int flags, long offset);
```

**32-Bit Syntax:**

```
long adx_tfseek (short session, unsigned short flags, long offset);
```

**Parameters:**

**session**

Session number of the open file.

**flags** Determines how to interpret the offset field.

**bits 0–7**

Reserved (must be 0)

**bits 8–9**

the file pointer is positioned

00 — relative to the beginning of the file

01 — relative to the current file pointer

10 — relative to the end of the file

**offset** Offset from the position is indicated by bits 8 and 9.

**bits 10–15**

Reserved (must be 0)

## Getting the Size of a File

This operation queries the size in bytes of an open disk file.

If the operation is successful, the size of the file is returned. A negative return value (-1) indicates that an error has occurred and that the error code and file number have been placed in global variables *adx\_errn* and *adx\_errf*, respectively.

**16-Bit Syntax:**

```
long adx_tsize(int session);
```

**32-Bit Syntax:**

```
long adx_tsize(short session);
```

**Parameters:**

**session**

Session number of the open file.

## File and Pipe Services

This section contains examples of C interfaces for keyed, non-keyed, and direct files. It also contains examples for pipes.

## Create a Non-POS File or Pipe

This function creates a non-keyed, non-POS file. If you create a file with a name that already exists, the existing file is erased before creating the new one. If you create a pipe with a pipe name that already exists, the create operation fails.

The application can create a file for reading or writing, or both, by setting the appropriate flag bits.

**Note:** This call does not create a controller pipe. To do so, you must use the call, *adx\_tcreate\_prs*

Disk file attributes pertaining to security are defined at creation. The most recently executed *adx\_tumask()* determines the file protection attributes.

If the operation is successful, the file is opened and a 2-byte session number is returned to be used for subsequent accesses to the open file. Otherwise, a negative value (-1) is returned and an error code is placed in global variable *adx\_errn*.

### 16-Bit Syntax:

```
int adx_tcreate_file (char _far *filename,unsigned int flags,
                    unsigned long filesize, unsigned int rectime,
                    unsigned int priority);
```

### 32-Bit Syntax:

```
short adx_tcreate_file (char *filename,unsigned short flags,
                      unsigned long filesize, unsigned short rectime,
                      unsigned short priority);
```

### Parameters:

#### filename

Valid file or pipe name or a previously defined logical name. The string can include the path specification. The maximum length of a file name in the controller is 25 bytes, including the NULL terminator. Pipe names must include the device identifier *pi*:

#### flags

##### bit 0

0 = No delete  
1 = Delete

**bit 1**    Reserved (must be 0)

##### bit 2

0 = No write access  
1 = Write access

##### bit 3

0 = No read access  
1 = Read access

##### bit 4

0 = Exclusive  
1 = Shared

##### bit 5

0 = Allow shared read and write if shared  
1 = Allow shared reads if shared

**bits 6–15**

Reserved (must be 0)

**filesize**

Size of the file in bytes. Use a size value of 0 for a file whose size is dynamic. For pipes, the size value specifies the size of the pipe buffer.

**recsize**

Record size. This is used by the read, write, and record lock functions to ensure that the requested action falls on a record boundary. Specify either 0 or 1 to prevent boundary checks from being performed.

**priority**

When this flag is 1 (true), I/O to this file has priority over other pending I/O requests. If it is 0, the file does not have priority. Valid only for disk files.

**Note:** Flag bit 0 specifies whether the application can delete the file before closing it. (In actual practice, the creator always has delete privileges.)

See the *4690 OS: Messages Guide* for a list of system error codes. See also “CPI Error Codes” on page 450 for a list of CAPI-generated error codes.

**Opening a File or Pipe**

This function opens an existing non-keyed file or terminal pipe. If the operation is successful, a 2-byte session number is returned for use in subsequent accesses to the file. Otherwise, a negative value is returned, and an error code is placed in global variable *adx\_errn*.

The application can open a file for reading or writing, or both, by setting flag bits 2 and 3 appropriately. Flag bit 0 specifies whether the application can delete the file before closing it.

**16-Bit Syntax:**

```
int adx_topen_file (char _far *filename, unsigned int flags,
                  unsigned int priority);
```

**32-Bit Syntax:**

```
short adx_topen_file (char *filename, unsigned short flags,
                    unsigned short priority);
```

**Parameters:****filename**

Valid file or pipe name or a previously defined logical name. The string must include the path specification when necessary. The maximum length of a file name in the controller is 25 bytes, including the NULL terminator. Pipe names must include the device identifier *pi:*.

**flags****bit 0**

0 = No delete

1 = Delete

**bit 1** Reserved**bit 2**

0 = No write access

1 = Write access

**bit 3**

0 = No read access

1 = Read access

**bit 4**

0 = Exclusive

1 = Shared

**bit 5**

0 = Allow shared read and write if shared

1 = Allow shared reads if shared

**bits 6–15**

Reserved (must be 0)

**priority**

When this flag is TRUE, I/O to this file has priority over other pending I/O requests. Valid only for disk files.

**Note:** This call cannot be used to open controller pipes.

See the *4690 OS: Messages Guide* for a list of system error codes.

## Closing a File or Pipe

This function closes a specified file or pipe. A zero value is returned if the operation is successful. Otherwise, a negative value (-1) is returned, an error code is placed in global variable *adx\_errn*, and the session number is placed in global variable *adx\_errf*.

**16-Bit Syntax:**

```
int adx_tclose_file (int session);
```

**32-Bit Syntax:**

```
short adx_tclose_file (short session);
```

**Parameters:**

**session**

Session number of file to be closed.

See the *4690 OS: Messages Guide* for a list of system error codes.

## Reading From a File or Pipe

This function reads a specified number of bytes from the specified file or pipe. The file pointer is updated on every read to the byte position immediately following the last byte read. If the operation is successful, the data is placed in the specified buffer and the number of bytes read is returned. If the number of bytes read is less than *buffsize*, an end-of-file was encountered.

A negative return value indicates that an error occurred. If an error occurs, the error code is placed in global variable *adx\_errn* and the session number is placed in global variable *adx\_errf*.

When reading a pipe, the system waits for enough data to be written to the pipe to satisfy the read request. Until this occurs, the application is suspended. Function *adx\_twait()*, described in “Waiting for Data in a Pipe Routing Services Pipe” on page 413, can be called before issuing a pipe read request. It suspends the application for a maximum specified time interval or until pipe and device data becomes available for reading.

### 16-Bit Syntax:

```
long adx_tread_file (int session,
                    unsigned int flags,
                    unsigned char _far *buffadr,
                    unsigned long buffsize,
                    long offset);
```

### 32-Bit Syntax:

```
long adx_tread_file (short session,
                    unsigned short flags,
                    unsigned char *buffadr,
                    unsigned long buffsize,
                    long offset);
```

### Parameters:

#### session

Session number of the open file to be read.

#### flags

##### bit 0–1

0 = Reserved (must be 0)

##### bit 2

0 = Normal read

1 = Non-destructive read. (The value of this bit has meaning only for pipes.)

##### bits 3–7

Reserved (must be 0)

##### bits 8–9

Determines how to interpret the offset field.

##### bits 10–15

Reserved (must be 0)

#### buffadr

Address of the buffer to contain the data that is read.

#### buffsize

Number of bytes to read.

**offset** For disk files this is the byte offset to begin reading, relative to the position indicated by *flag* bits 8 and 9. Negative offsets are allowed. *offset* should be set to 0 for pipes.

See the *4690 OS: Messages Guide* for a list of system error codes.

## Writing to a File or Pipe

This function writes a specified number of bytes to a non-keyed file or pipe. The file pointer is updated on every write to the byte position immediately following the last byte written. If the operation is successful, the number of bytes written is returned. If the number of bytes written is less than *buffsize*, an end-of-media was detected.

A negative return value indicates that an error occurred. If an error occurs, the error code is placed in global variable *adx\_errn* and the session number is placed in global variable *adx\_errf*.

When writing to a pipe, if the buffer is full, the operation waits for enough data to be read from the buffer to make room for the write request.

### 16-Bit Syntax:

```
long adx_twrite_file(int session,
                    unsigned int flags,
                    unsigned char _far *buffadr,
                    unsigned long buffsize,
                    long offset);
```

### 32-Bit Syntax:

```
long adx_twrite_file(short session,
                    unsigned short flags,
                    unsigned char *buffadr,
                    unsigned long buffsize,
                    long offset);
```

### Parameters:

#### session

Session number of the open file.

#### flags

**bit 0** determines internal buffering.

0 — flush buffers after WRITE. This forces data to the media to ensure data integrity. If this is a zero length request, the media is updated with any pending writes.

1 — allow optimized internal buffering.

#### bits 2–7

Reserved (must be 0)

#### bits 8–9

Determines how to interpret the offset field.

00 — relative to beginning of the file

01 — relative to file pointer (disk file only)

10 — relative to end of file (disk file only)

#### bits 10–15

Reserved (must be 0)

#### buffadr

Address of the buffer that contains the data that is to be written.

#### buffsize

Number of bytes to write.

**offset** For disk files this is the byte offset to begin writing, relative to the position indicated by *flag* bits 8 and 9. This should be 0 for pipes.

See the *4690 OS: Messages Guide* for a list of system error codes.

## Writing a String Array to a Sequential File

This operation is a special disk write that efficiently writes a series of strings to disk. It is valid only for sequential files. Each string can be no longer than 508 bytes.

Write matrix is performed by a driver designed to handle one-dimensional BASIC string arrays, which are organized as an array of far pointers to BASIC-style strings. The following type definition defines a BASIC-style string. The first two bytes contain the length of the string, and that a null terminator is not required.

### **16-Bit Syntax:**

```
long adx_twrite_matrix(int sessnum,  
                      BSTRING_t_far * _far * element,  
                      unsigned int nelem);
```

### **32-Bit Syntax:**

```
long adx_twrite_matrix(short sessnum,  
                      BSTRING_t * * element,  
                      unsigned short nelem);
```

#### **Parameters:**

##### **session**

Session number of the open file.

##### **element**

Address of an array of pointers to BASIC-style strings.

**nelem** Number of elements to write.

### **Open a Keyed File**

This function opens an existing keyed file. If the open is successful, the function return value is a 2-byte session number, which the application uses for subsequent accesses to the file. If not successful, the function return value is negative (-1) and an error code is placed in global variable *adx\_errn*.

The application can open a file for reading or writing, or both, by setting flag bits 2 and 3 appropriately. Flag bit 0 specifies whether the application can delete the file before closing it.

### **16-Bit Syntax:**

```
int adx_topen_keyed(char _far *filename,  
                   unsigned int flags,  
                   unsigned int priority);
```

### **32-Bit Syntax:**

```
short adx_topen_keyed(char *filename,  
                     unsigned short flags,  
                     unsigned short priority);
```

#### **Parameters:**

##### **filename**

Name of the file or a previously defined logical name. The string must include the path specification when necessary. The maximum length is 25 bytes, including the NULL terminator.

##### **flags**

###### **bit 0**

0 = No delete  
1 = Delete

**bit 1** Reserved (must be 0)

###### **bit 2**

0 = No write access  
1 = Write access

###### **bit 3**

0 = No read access  
1 = Read access

**bit 4**

0 = Exclusive  
1 = Shared

**bit 5**

0 = Allow shared read and write if shared  
1 = Allow shared reads if shared

**bits 6–15**

Reserved (must be 0)

**priority**

1 = Disk accesses for this file have priority over other pending disk requests.  
0 = Disk accesses for this file do not have priority.

See the *4690 OS: Messages Guide* for a list of system error codes.

**Close a Keyed File**

This function closes a keyed file. A zero return value indicates that the operation was successful. A negative return value indicates that an error has occurred. In this case, an error code is placed in global variable *adx\_errn*, and the session number is placed in global variable *adx\_errf*.

**16-Bit Syntax:**

```
int adx_tclose_keyed (int session,
                    unsigned int option);
```

**32-Bit Syntax:**

```
short adx_tclose_keyed (short session,
                      unsigned short option);
```

**Parameters:****session**

Session number of file to be closed.

**option**

0 = Close  
1 = Zero and close (4690 BASIC's "CLOSE QUIESCE")

See the *4690 OS: Messages Guide* for a list of system error codes.

**Read a Keyed File**

This function reads a specified record from an open keyed file. If the operation is successful, the record is placed in the caller's buffer, and the number of bytes read is returned. A negative return value indicates that an error has occurred, and an error code is placed in global variable *adx\_errn* and the session number is placed in global variable *adx\_errf*.

**16-Bit Syntax:**

```
long adx_tread_keyed (int session, unsigned int option,
                    unsigned char _far *buffadr,
                    unsigned long buffsize,
                    unsigned long keylen);
```

**32-Bit Syntax:**

```
long adx_tread_keyed (short session, unsigned short option,
                    unsigned char *buffadr,
                    unsigned long buffsize,
                    unsigned long keylen);
```



**Parameters:****session**

Session number of the open file to be read.

**option**

0 = Read without locking

1 = Lock the record before reading it. If the record is not available, the application must wait for it to become available. This lock is effective only against controller applications.

**buffadr**

Address of the buffer to contain the record that is read. The key for the record to be read must be in the buffer at offset 0.

**buffsize**

Number of bytes to read. This must equal the record size of the file, but cannot exceed 508 bytes.

**keylen**

Length of the key string in the buffer.

See the *4690 OS: Messages Guide* for a list of system error codes.

**Delete a Keyed File Record**

This function deletes a specified record in a keyed file. If the operation is successful, zero is returned. If an error occurs, a negative value is returned (-1), an error code is placed in global variable *adx\_errn*, and the session number is placed in global variable *adx\_errf*.

**16-Bit Syntax:**

```
long adx_tdelete_krec(int session,
                     unsigned char _far *buffadr,
                     unsigned long keylen);
```

**32-Bit Syntax:**

```
long adx_tdelete_krec(short session,
                     unsigned char *buffadr,
                     unsigned long keylen);
```

**Parameters:****session**

Session number of the open file from which the record is to be deleted.

**buffadr**

Address of the buffer containing the key of the record to be deleted. The key must begin at offset 0 in this buffer.

**keylen**

Length of key string.

See the *4690 OS: Messages Guide* for a list of system error codes.

**Write to a Keyed File**

This function writes data to a specified record in a keyed file. If the operation is successful, the number of bytes written is returned. If an error occurs, a negative value is returned, the error code is placed in global variable *adx\_errn*, and the session number is placed in global variable *adx\_errf*.

**16-Bit Syntax:**

```
long adx_twrite_keyed(int session, unsigned int option,
                     unsigned char _far *buffadr,
                     unsigned long buffsize);
```

### 32-Bit Syntax:

```
long adx_twrite_keyed(short session, unsigned short option,  
                    unsigned char *buffadr,  
                    unsigned long bufsize);
```

#### Parameters:

##### session

Session number of the open file in which the data is to be written.

##### option

0 = Do not unlock the record

1 = Unlock the record after writing it

##### buffadr

Address of the buffer that contains the data that is to be written. The key for the record to be written must be in the buffer starting at offset 0

##### bufsize

Number of bytes to write. This must equal the record size of the file, but cannot exceed 508 bytes.

See the *4690 OS: Messages Guide* for a list of system error codes.

---

## Pipe Routing Services

| Applications in the 469x, SurePOS 300/700 or TCxWave 6140 Series terminal write data to pipes that  
| already exist in the controller. This is accomplished by using a special Pipe Routing Services (PRS) driver  
| in the terminal. The application opens the driver and then uses a combination of the driver session number  
| and a pipe address to perform the writes.

| Pipes from which the application in the 469x, SurePOS 300/700 or TCxWave 6140 Series terminal can  
| read are created by the application executing in the 469x/4800-7xx/4900-7x5/4800-350/6140-100 terminal.  
| These pipes also physically exist in the controller, but it is not necessary to open the PRS driver to create  
| or read from them.

## Create a Pipe Routing Services Pipe

This function creates a PRS pipe in the controller that the terminal application can read. A controller application or another terminal application can write to this pipe. If the pipe is created successfully, the function returns a positive 2-byte session number. This session number is used for all subsequent operations on that pipe until the pipe is closed. A negative return value (-1) indicates that an error has occurred and the error code placed in global variable *adx\_erm*.

### 16-Bit Syntax

```
int adx_tcreate_prs (char pipeid,  
                   unsigned int bufsize);
```

### 32-Bit Syntax

```
short adx_tcreate_prs (char pipeid,  
                     unsigned short bufsize);
```

#### Parameters:

**pipeid** Alphabetic character that uniquely identifies this pipe. Valid values are ASCII A through Z. Lowercase is forced to upper case, so only 26 unique pipe IDs are possible.

##### bufsize

Size of the pipe in bytes. The maximum size is 240 bytes. If a size greater than 240 is specified, the size is limited to 240 bytes, and no error is returned.

See the *4690 OS: Messages Guide* for a list of system error codes.

## Waiting for Data in a Pipe Routing Services Pipe

This function permits the application to wait for data to become available from one or more input devices and pipes. See “Waiting for Data” on page 434 for details.

## Reading a Pipe Routing Services Pipe

This function reads a specified number of bytes from an open PRS pipe. If the operation is successful, the data is placed in the caller's buffer and the number of bytes read is returned. A negative return value (-1) indicates that an error has occurred, in which case the error code placed in global variable *adx\_errn*, and the session number is placed in global variable *adx\_errf*.

### 16-Bit Syntax

```
int adx_tread_prs (int session,
                  unsigned char _far *buffadr,
                  unsigned int buffsize);
```

### 32-Bit Syntax

```
short adx_tread_prs (short session,
                    unsigned char *buffadr,
                    unsigned short buffsize);
```

#### Parameters:

##### **session**

Session number of the open pipe.

##### **buffadr**

Address of the buffer to receive the data that is read.

##### **buffsize**

Number of bytes requested. The maximum PRS message size is 120 bytes. An attempt to read more than 120 bytes results in an error.

See the *4690 OS: Messages Guide* for a list of system error codes.

## Close a Pipe Routing Services Pipe

This function closes an existing PRS pipe. The pipe must have been created by the program with *adx\_tcreate\_prs()*. If the operation is successful, the function returns 0. A negative return (-1) indicates that an error has occurred, in which case the error code is placed in global variable *adx\_errn*, and the session number placed in global variable *adx\_errf*.

### 16-Bit Syntax

```
int adx_tclose_prs (int session);
```

### 32-Bit Syntax

```
short adx_tclose_prs (short session);
```

#### Parameters:

##### **session**

Session number of the file to be closed.

## Open the Pipe Routing Services Driver

This function opens the PRS driver in preparation for sending data using a pipe to applications in other terminals or controllers. If the operation is successful, the driver is opened, and a positive 2-byte session

number is returned. This session number is used for all subsequent write operations to PRS pipes and is valid until the driver is closed. A negative return (-1) indicates that an error occurred, in which case the error code is placed in global variable *adx\_errn*.

### 16-Bit Syntax

```
int adx_tinit_prs(void);
```

### 32-Bit Syntax

```
short adx_tinit_prs(void);
```

See the *4690 OS: Messages Guide* for a list of system error codes.

## Writing to a Pipe Routing Services Pipe

This function writes a specified number of bytes to an open PRS pipe. If the operation is successful, the number of bytes written is returned. A negative return value (-1) indicates that an error has occurred, in which case the error code is placed in global variable *adx\_errn*, and the session number is placed in global variable *adx\_errf*.

### 16-Bit Syntax

```
int adx_twrite_prs (int session, char _far *dest,  
                   unsigned char _far *buffadr,  
                   unsigned int bufsize);
```

### 32-Bit Syntax

```
short adx_twrite_prs (short session, char *dest,  
                     unsigned char *buffadr,  
                     unsigned short bufsize);
```

#### Parameters:

##### session

Session number of the PRS driver returned by the *adx\_tinit\_prs*.

##### dest

Destination address

This argument is a null-terminated string containing four characters in the form *aaaw* where:

- *aaa* indicates which terminal or store controller to send this data to. For terminals, it is the terminal number in ASCII. For store controllers, it consists of *0xy*, where *x* and *y* are characters between C and Z. Therefore, for the store controller, *aaa* can range from 0CC to 0ZZ.

There are also two special values of *aaa* for store controllers: 0AA and 0BB. Use 0AA when the destination is the Master store controller. Use 0BB when the destination is the controller to which the terminal is attached. These can be thought of as logical destinations. Pipe Routing Services performs the translations of 0AA and 0BB so that your application does not have to actually know the real store controller IDs.

- *w* is the PRS pipe ID, which is the ID used by an application in the destination terminal or controller to create the pipe.

See “Pipe Routing Services” on page 287 further information.

##### buffadr

Address of the buffer that contains the data that is to be written.

##### bufsize

Number of bytes to write. The maximum permitted is 512 bytes.

## Conditional Write to a Pipe Routing Services Pipe

This function is identical in every way to the *adx\_twrite\_prs()* function with one exception: if the destination pipe is full or does not have enough room left to contain the entire message being written, the write does

not wait for room to become available. Instead, the application is given control immediately with a -1 return code, and `adx_errn` is also set to -1. At this point, the application can make a decision to either discard the data, or to retry the write at a later time.

The intended use for the conditional pipe write is for situations where it is undesirable for an application to wait for extended periods of time.

The function is `adx_tcwrite_prs()`. See “Writing to a Pipe Routing Services Pipe” on page 414 for information on the syntax and parameters.

## Close the Pipe Routing Services Driver

This function closes the PRS driver. If the operation is successful, the driver is closed and the function returns zero. A negative return indicates that an error occurred, in which case the error code is placed in global variable `adx_errn`.

### 16-Bit Syntax

```
int adx_tterm_prs(int session);
```

### 32-Bit Syntax

```
short adx_tterm_prs(short session);
```

#### Parameters:

##### session

Session number of the open PRS driver.

See the *4690 OS: Messages Guide* for a list of system error codes.

---

## CPI Device Interface

The CPI device interface is similar to the 4680 BASIC interface. Where possible, function and argument names are the same as or similar to the corresponding BASIC names. In most cases, the descriptions of BASIC functions and arguments apply equally to the CPI. Differences are noted where they occur. For more information on point-of-sale devices and of the I/O services provided by this API, see the *4680 BASIC: Language Reference*.

4680 BASIC provides a high level of I/O services such as data buffering and formatting, and sequential, direct or random file access. The CPI operates at a lower level. It is the responsibility of the C application to format and buffer data, specify and remember file offsets, specify buffer addresses and lengths.

This section lists each BASIC I/O routine that accesses point-of-sale devices and also contains the prototype and description of the corresponding C Interface function.

The argument *session* is used by most of the functions. This is the 2-byte session number returned by the C Interface when the device is opened.

As a general rule, a negative return value (-1) from a C Interface function indicates that an error has occurred in its execution. The error code is placed in global variable `adx_errn` and the session number, if applicable, is placed in global variable `adx_errf`. The application can reference these variables to analyze the error. See the *4690 OS: Messages Guide* for a list of system error codes. Error codes generated by the C Interface are listed in “CPI Error Codes” on page 450.

## Open a Device

This function opens an I/O device.

## 16-Bit Syntax

```
int adx_topen_device(char _far *name);
```

## 32-Bit Syntax

```
short adx_topen_device(char *name);
```

### Parameters:

The device name must be one of the following.

#### “ANDISPLAY”

First alphanumeric display

#### “ANDISPLAY2”

Second alphanumeric display

#### “ANDISPLAY3”

Third alphanumeric display

#### “CDRAWER”

Cash drawer

#### “COIN”

Coin return

#### “MSR”

Magnetic stripe reader

#### “CR” Printer customer receipt station

#### “SJ” Printer summary journal station

#### “DI” Printer document insert station

#### “SCALE”

Scale driver

#### “SDISPLAY”

Shopper display

#### “TONE”

Keyboard tone driver

#### “TOTRET”

Totals Retention (nonvolatile RAM)

#### “VDISPLAY”

First video display

#### “VDISPLAY2”

Second video display

A positive return value from `adx_topen_device()` is the 2-byte session number that the application uses to access the device, i.e. read, write, etc. A negative return value (-1) indicates that an error has occurred and an error code placed in global variable `adx_errn`. See the *4690 OS: Messages Guide* for a list of system error codes.

## Open the I/O Processor

This open function, which is unique to the I/O Processor, passes the names of state tables to the system on the open call. This eliminates a separate function call to load the state tables.

## 16-Bit Syntax

```
int adx_topen_ioproc(char _far *istbl,  
                    char _far *fmttbl,  
                    char _far *modtbl,  
                    unsigned int buffnum);
```

## 32-Bit Syntax

```
short adx_topen_ioproc(char *istbl,  
                      char *fmttbl,  
                      char *modtbl,  
                      unsigned short buffnum);
```

### Parameters:

**istbl**     Name of the input sequence table. This is required.

**fmttbl**  
           Name of the label format table.

**modtbl**  
           Name of the modulo check table.

**buffnum**  
           Maximum number of items on the input sequence queue.

The table names are the names of files on the controller. The maximum name length is 41 characters, including the null terminator. If a table name is omitted, the corresponding argument must be a null pointer: (*char \_far \**)0L.

If the operation is successful, the function return value is the positive 2-byte session number that is used to access the device. A negative return value indicates that an error has occurred and an error code placed in global variable *adx\_errn*. See the *4690 OS: Messages Guide* for a list of system error codes.

## Open a Serial I/O Session

This function opens a serial I/O session. These arguments are as documented in the *4680 BASIC: Language Reference* with the following exceptions:

- *parity* must be E, O or N.
- For 4690 OS Version 5, *buffsize* can be a maximum of 1024 bytes for serial devices attached to RS485 ports and a maximum of 16K bytes for serial devices attached to RS232 ports.
- For 4690 OS Version 5, *speed* can be a maximum of 115.2K bits per second (bps) for serial devices attached to RS232 ports and a maximum of 9600 bps for serial devices attached to RS485 ports. The allowed values for *speed* are 110, 1200, 2400, 4800, 9600, 19200, 14400, 28800, 38400, 57600, and 1152 (which is used to specify 115200, because *speed* is a 2-byte integer value and 115200 cannot be contained in the 2 bytes).

## 16-Bit Syntax

```
int adx_topen_serial(unsigned int port,  
                    unsigned int speed,  
                    unsigned char parity,  
                    unsigned int data,  
                    unsigned int stop,  
                    unsigned int buffsize);
```

## 32-Bit Syntax

```
short adx_topen_serial(unsigned short port,
                      unsigned short speed,
                      unsigned char parity,
                      unsigned short data,
                      unsigned short stop,
                      unsigned short bufsize);
```

If the operation is successful, the function return value is the positive 2-byte session number that is used to access the device. A negative return value (-1) indicates that an error has occurred and an error code placed in global variable *adx\_errn*. See the *4690 OS: Messages Guide* for a list of system error codes.

## Close a Device

This function closes a device.

### 16-Bit Syntax

```
int adx_tclose_device(int session);
```

### 32-Bit Syntax

```
short adx_tclose_device(short session);
```

#### Parameters:

##### session

Session number of the device.

The return value is zero if the operation is successful. Otherwise, a negative value (-1) is returned and the error code and the session number are placed in global variables *adx\_errn* and *adx\_errf*. See the *4690 OS: Messages Guide* for a list of system error codes.

## Position the Cursor

This function positions the cursor.

### 16-Bit Syntax

```
int adx_tlocate(int session, unsigned int row,
               unsigned int column,
               unsigned int cursor_on);
```

### 32-Bit Syntax

```
short adx_tlocate(short session, unsigned short row,
                  unsigned short column,
                  unsigned short cursor_on);
```

#### Parameters:

##### session

Session number of the display.

**row** For an alphanumeric or operator display, this parameter must be either 1 or 2. For a video display, the valid range is defined by the current display format. (The display format is determined at configuration, and can be modified by means of *adx\_putlong*.)

##### column

For an alphanumeric or operator display, this parameter must be in the range of 1 to 20. For a video display, the valid range is determined by the current display format.



### **cursor\_on**

This parameter specifies whether the cursor is displayed. If this parameter is 1, the cursor is visible. If this parameter is any other value, the cursor is not visible. This parameter is ignored for the alphanumeric display.

The return value is zero if the operation is successful. Otherwise, a negative value is returned and the error code and the session number are placed in global variables *adx\_errn* and *adx\_errf*. See the *4690 OS: Messages Guide* for a list of system error codes.

## **Clear the Display**

This function clears the display.

### **16-Bit Syntax**

```
int adx_tclears(int session, unsigned int reset);
```

### **32-Bit Syntax**

```
short adx_tclears(short session, unsigned short reset);
```

#### **Parameters:**

##### **session**

Session number of the display.

**reset** 0, if the cursor is to remain at the current location; 1, if it is to return to the HOME position following the clear screen operation.

The return value is zero if the operation is successful. Otherwise, a negative value (-1) is returned and the error code and the session number are placed in global variables *adx\_errn* and *adx\_errf*. See the *4690 OS: Messages Guide* for a list of system error codes.

## **Get Device Status Bytes**

This version of GETLONG differs from BASIC in that the 4 status bytes are placed in the buffer. The GETLONG function of 4680 BASIC returns these bytes as the function return value.

### **16-Bit Syntax**

```
int adx_tgetlong(int session,
                 unsigned long _far *status);
```

### **32-Bit Syntax**

```
short adx_tgetlong(short session,
                   unsigned long *status);
```

The significance of the status bits varies by device. See Chapter 3, "Programming Terminal I/O Devices," on page 39 and see the *4680 BASIC: Language Reference* for more information.

The return value is zero if the operation is successful. Otherwise, a negative value (-1) is returned and the error code and the session number are placed in global variables *adx\_errn* and *adx\_errf*. See the *4690 OS: Messages Guide* for a list of system error codes.

## **Set Device Status Bytes**

This function sets the device status bytes.

### **16-Bit Syntax**

```
int adx_tputlong(int session, unsigned long data);
```

## 32-Bit Syntax

```
short adx_tputlong(short session, unsigned long data);
```

See the *4680 BASIC: Language Reference* for the correct format of the *data* parameter for each device.

**Note:** Changes to the shopper display guidance lights do not take effect until the next write to the shopper display.

The return value is zero if the operation is successful. Otherwise, a negative value (-1) is returned and the error code and the session number are placed in global variables *adx\_errm* and *adx\_errf*. See the *4690 OS: Messages Guide* for a list of system error codes.

## Load State Tables

This function is not implemented separately in the C Interface. Its purpose in BASIC is to specify the names of the I/O Processor state tables prior to opening the I/O Processor. In the C API, the names of the state tables are arguments on the *adx\_topen\_ioproc()* call.

## Lock a Device

This function locks a device and is valid only for the MSR and the I/O Processor.

### 16-Bit Syntax

```
int adx_tlockdev(int session, unsigned int purge);
```

### 32-Bit Syntax

```
short adx_tlockdev(short session, unsigned short purge);
```

The *purge* flag has meaning only for the I/O Processor. It can be either 0 (false) or 1 (true). If true, all data queued to your application at the time of the lock is discarded.

The return value is zero if the operation is successful. Otherwise, a negative value (-1) is returned and the error code and the session number are placed in global variables *adx\_errm* and *adx\_errf*. See the *4690 OS: Messages Guide* for a list of system error codes.

## Unlock a Device

This function unlocks a device and is valid only for the MSR and the I/O Processor.

### 16-Bit Syntax

```
int adx_tunlockdev(int session,
                   unsigned int newstate,
                   unsigned int priority);
```

### 32-bit Syntax

```
short adx_tunlockdev(short session,
                     unsigned short newstate,
                     unsigned short priority);
```

The *priority* flag can be either 0 (false) or 1 (true).

The *newstate* argument is meaningful only to the I/O Processor. A *newstate* of 0 leaves the I/O processor in its current state.

The return value is zero if the operation is successful. Otherwise, a negative value is returned and the error code and the session number are placed in global variables *adx\_errm* and *adx\_errf*. Refer to the *4690 OS: Messages Guide* for a list of system error codes.

## Define an Asynchronous Error Handler

This operation designates a function to be called when an asynchronous run-time error occurs. Asynchronous errors are associated with writes to the terminal printer, serial I/O driver, and coin dispenser.

### 16-Bit Syntax

```
int on_async_err_call(ASYNC_ERROR_HANDLER_t
                     error_handler,
                     ASYNC_ERROR_HANDLER_t
                     _far *old_value);
```

where:

```
typedef void _far (*ASYNC_ERROR_HANDLER_t)
                 (int,
                  unsigned long,
                  unsigned int*,
                  char*);
```

### 32-Bit Syntax

```
short on_async_err_call(ASYNC_ERROR_HANDLER_t
                       error_handler,
                       ASYNC_ERROR_HANDLER_t
                       *old_value);
```

where:

```
typedef void (_Optlink *ASYNC_ERROR_HANDLER_t)
            (short,
             unsigned long,
             unsigned short*,
             char*);
```

#### Parameters:

##### error\_handler

Pointer to the application's asynchronous error handling routine. This routine is called by the CPI when an error associated with an asynchronous event occurs.

##### old\_value

Address of a buffer where the API is to return the previous async error handler address. It is returned to the calling routine. If there has been no previous *on\_async\_err\_call*, this value is NULL. If this address is not needed, specify NULL for this parameter. This address is available to allow restoration of a previous *on\_async\_err\_call* address in the case of nested calls.

The 16-bit application's asynchronous error handling routine must be in the following format:

```
void async_error_handler(int session,
                        unsigned long errcode,
                        unsigned int *retryflag,
                        char *overlaystr);
```

The 32-bit application's asynchronous error handling routine must be in the following format:

```
void _Optlink async_error_handler(short session,
                                unsigned long errcode,
                                unsigned short *retryflag,
                                char *overlaystr);
```

#### Parameters:

##### session

Input parameter identifying the device that experienced the error.

**errcode**

Input parameter identifying the error that occurred.

**retryflag**

Specifies if the failing I/O operation is to be attempted again. A value of 0 for the retry flag informs the run-time library not to retry the operation. A value of -1 indicates the operation is to be retried.

**overlaystr**

Specifies if any data are to overlay part of the failing print line. Also specified is the offset where these characters are to be printed. If you do not want to overlay part of the print line, or if the failing I/O is not associated with the printer, this variable must be a null string. The format of the string data for this variable (if not null) is:

- 2 bytes indicating the offset to use followed by
- The overlay string (1–38 characters)

If the application supplies an overlay string, it copies the string data into the buffer pointed to by *overlaystr*.

**Note:** When copying data into the *overlaystr* buffer, do not exceed the size of the buffer for the printer model and station being used. Refer to “Write to a Device” on page 425 for maximum permitted buffer lengths.

**Note:** Global variables, *adx\_errn* and *adx\_errf*, are not contained data pertaining to the error that caused the asynchronous error handler to be invoked. The asynchronous error handler should reference its input parameters to determine the nature of the error.

The asynchronous error handler is logically part of the CPI software interrupt (SWI) routine, which can pre-empt control at any time and interrupt execution of mainline code. SWIs themselves are non-interruptible.

Any function or library routine that is called both from a SWI and from mainline code must be reentrant. This means that it must not modify static data. Similarly, data that can be modified by a SWI should not be read or modified in the mainline code unless SWIs are disabled while the mainline code is operating on the data.

SWIs can be disabled and enabled using functions *adx\_tdisable()* and *adx\_tenable()*, documented in “Miscellaneous Routines” on page 428.

Be sure to re-enable SWIs after having disabled them. SWIs should remain disabled for as short a time as possible. If SWIs remain disabled, the application runs out of event bits.

Writes to the terminal printer, serial I/O driver and coin dispenser use asynchronous I/O and require SWIs to run. When a write is done to one of these devices, control is returned immediately to the application, and an event bit is allocated for the write operation. Later, when the write is logically complete, the SWI is scheduled to run.

The SWI performs two operations:

- It checks for an error, and, if one occurred, it calls the application's asynchronous error handler.
- It frees the event bit that was allocated for this write.

If SWI execution is enabled, the SWI runs as soon as it is scheduled. Otherwise the SWI operation is delayed until SWIs are re-enabled, and the event bit remains allocated.

Event bits are allocated from a pool of 31 per application. Therefore, if a program logic bug causes the SWI operation to be left disabled, the application continues to run, and SWIs stack up while waiting to be

run. Eventually, the application runs out of event bits and blocks all further asynchronous operations by the operating system on behalf of the application. The system error code associated with this condition is 80004186.

In 32-bit applications, asynchronous error SWIs run on the same thread that initiated the asynchronous operation (write). As in the 16-bit interface, other SWIs for that thread are non-interruptible. However, other threads continue to run (and can be interrupted by SWIs themselves). The pool of 32 SWIs is unique per thread within a process.

## Set Totals Retention Offset

This function sets the Totals Retention offset and is valid only for the Totals Retention (nonvolatile RAM) device. The offset of the first byte in Totals Retention is 1, not 0.

### 16-Bit Syntax

```
int adx_tpoint(int session, unsigned long offset);
```

### 32-Bit Syntax

```
short adx_tpoint(short session, unsigned long offset);
```

The return value is zero if the operation is successful. Otherwise, a negative value is returned and the error code and the session number are placed in global variables *adx\_ern* and *adx\_errf*. See the *4690 OS: Messages Guide* for a list of system error codes.

## Get Totals Retention Offset

This function reports the offset of the beginning of the last record read or written to Totals Retention. This function is valid only for the Totals Retention (nonvolatile RAM) device.

### 16-Bit Syntax

```
int adx_tptrrtn(int session, unsigned long_far *offset);
```

### 32-Bit Syntax

```
short adx_tptrrtn(short session, unsigned long *offset);
```

This version differs from BASIC in that the offset value is placed in the buffer. The 4680 BASIC function returns the offset value as the function return value.

The return value is zero if the operation is successful. Otherwise, a negative value (-1) is returned and the error code and the session number are placed in global variables *adx\_ern* and *adx\_errf*. See the *4690 OS: Messages Guide* for a list of system error codes.

## Read from a Device

This function receives data from a device.

### 16-Bit Syntax

```
long adx_tread_device(int session,
                     unsigned char _far *buffer,
                     unsigned long length);
```

### 32-bit Syntax

```
long adx_tread_device(short session,
                     unsigned char *buffer,
                     unsigned long length);
```

If the operation is successful, the function returns the number of bytes read from the device. Otherwise, a negative value (-1) is returned and the error code and the session number are placed in global variables *adx\_ern* and *adx\_errf*. See the *4690 OS: Messages Guide* for a list of system error codes.

The following table shows the maximum read buffer sizes for the devices.

Device	Buffer Size
ANDISPLAY	40
ANDISPLAY2	40
ANDISPLAY3	40
CDRAWER	N/A
COIN	N/A
IOPROC	The buffer size must be large enough to contain the largest input sequence that can be generated according to the Input State Table.
MSR	144  286 bytes for the 4693 Triple-track MSR (4690 OS V2 or higher) or 144 bytes for all other MSR types. See Chapter 3, "Programming Terminal I/O Devices," on page 39 for the format of data read from the MSR using the <i>adx_tread_device</i> and the <i>adx_tgetlong</i> functions.
CR	N/A
SJ	N/A
DI	The buffer must be large enough to accommodate a 65-byte read request.
SCALE	4
SDISPLAY	12
SERIAL I/O	Specified by the application when opened.
TONE	N/A
TOTRET	1016
VDISPLAY	2000
VDISPLAY2	2000

The format of data received from the POS devices is the same whether it is read through the C Interface or through the 4680 BASIC API.

**Note:** Data formats for the POS devices are fully documented in the *4680 BASIC: Language Reference* and in Chapter 3, "Programming Terminal I/O Devices," on page 39.

Reading from the I/O Processor is analogous to using the 4680 BASIC READ # LINE statement. That is, the entire contents of the input sequence buffer is written into the caller's buffer. This data consists of a 14-byte header field followed by up to 10 function code/data fields. The format of the data received from the IOPROC is multiple fields enclosed in double quotes and separated by commas. The last field is followed by a carriage return (CR).

An empty field is indicated by a comma unless it is the final field, in which case it retains the double quotes, for example,

```
"header","field1","field2",,"field4","
```

(Fields 3 and 5 are empty.)

When the application issues a read request to the I/O Processor, MSR or Serial I/O, it is suspended until data becomes available. Function *adx\_twait()* can be called before issuing a read request to these devices. It suspends the application for a specified time interval or until data becomes available for reading.

A read of valid data from the MSR leaves it in the locked state. It must be unlocked before attempting to read it again.

When reading the MICR at the DI station, the *length* parameter and the buffer size must be at least 65 bytes, although less than 65 bytes of data can be returned. A read request of less than 65 bytes has unpredictable results.

## Read Direct from Totals Retention

This function allows a direct read to the Totals Retention device and is valid only for the Totals Retention (nonvolatile RAM) device. This function allows a read of up to 1020 bytes of totals retention data rather than the 1016 bytes allowed using the *adx\_read\_device* function.

### 16-Bit Syntax

```
long adx_tr_read_direct(int session,
                        unsigned char _far *buffer,
                        unsigned long length,
                        unsigned long offset);
```

### 32-Bit Syntax

```
long adx_tr_read_direct(short session,
                        unsigned char *buffer,
                        unsigned long length,
                        unsigned long offset);
```

If the operation is successful, the function returns the number of bytes read. Otherwise, a negative value (-1) is returned and the error code and the session number are placed in global variables

## Wait for Event Completion

This service is provided by the *adx\_twait()* and *adx\_timer\_set()* functions described in “Waiting for Data” on page 434 and “Suspending Processing for Indicated Duration” on page 434.

## Write to a Device

This function enables you to write to a device.

### 16-Bit Syntax

```
long adx_twrite_device(int session,
                       unsigned char _far *buffer,
                       unsigned long length);
```

### 32-Bit Syntax

```
long adx_twrite_device(short session,
                       unsigned char *buffer,
                       unsigned long length);
```

#### Parameters:

##### session

Session number returned when the device was opened.

##### buffer

Pointer to the write data.

**length**

Length of data to write.

**Note:** Data formats for the POS devices are fully documented in the *4680 BASIC: Language Reference* and in Chapter 3, “Programming Terminal I/O Devices,” on page 39.

If the operation is successful, the function returns the number of bytes written to a synchronous device, or returns 0 if writing to an asynchronous device (a coin, printer, or serial I/O). If the operation fails, a negative value (-1) is returned and the error code and the session number are placed in global variables *adx\_errn* and *adx\_errf*. See the *4690 OS: Messages Guide* for a list of system error codes.

Table 51 shows the maximum values of the length parameter for each device. It is very important not to exceed these lengths.

*Table 51. Maximum Length Values for Devices*

Device	Maximum Value
ANDISPLAY	40
ANDISPLAY2	40
ANDISPLAY3	40
CDRAWER	1
COIN	4
IOPROC	A write to the I/O Processor writes data that has been read from the I/O Processor and modified by the application. It is good practice to return the same buffer and the same buffer length that were used on the read.
MSR	N/A
CR	39 (Model 1 or 2) or 76 (Model 3)
SJ	39 (Model 1 or 2) or 76 (Model 3)
DI	39 (Model 1 or 2) or 116 (Model 3)
SCALE	N/A
SDISPLAY	12
SERIAL I/O	Specified by the application when opened using <i>adx_topen_serial</i>
TONE	7
TOTRET	1016
VDISPLAY	2000
VDISPLAY2	2000

**Note:** The length of the data written to any of the printer stations must be at least 39 bytes. If it is less, the printer driver returns an invalid data error.

For a Model 1 or 2 printer, exactly 39 bytes of data must be sent to the printer. The first 38 bytes are printable characters, padded with blanks as necessary. The binary value in byte 39 is interpreted as the number of line feeds to advance the paper.

Because control characters can be embedded in the print data sent to the Model 3 printer, the quantity of data sent to the Model 3 printer driver for a single write can exceed 39 bytes. The final byte of data is interpreted as either the number of line feeds or the number of motor steps to advance the paper according to the current setting of this option.



The application program should ensure that the number of characters specified in a given font on the Model 3 printer can be printed on the station being used. The printer driver returns an error if the line width exceeds the width of the station.

To execute a receipt paper cut on the Model 3 printer, format the print line as follows: the first two bytes of data are decimal 27 and decimal 80, the next 36 bytes are blanks, and the final byte (at offset 38) is 0.

For more information on the Model 3 Printer, see “Printer Driver Model 3 or 4/4A” on page 106.

## Write Direct to Totals Retention

This function allows a direct write to the Totals Retention Device and is valid only for the Totals Retention (nonvolatile RAM) device.

### 16-Bit Syntax

```
long adx_tr_write_direct(int session,
                        unsigned char _far *buffer,
                        unsigned long length,
                        unsigned long offset);
```

### 32-Bit Syntax

```
long adx_tr_write_direct(short session,
                        unsigned char *buffer,
                        unsigned long length,
                        unsigned long offset);
```

This function allows a write of up to 1020 bytes rather than the 1016 bytes allowed by the *adx\_twrite\_dev* function.

If the operation is successful, the function returns the number of bytes written to the device. Otherwise, a negative value (-1) is returned and the error code and the session number are placed in global variables *adx\_errn* and *adx\_errf*. See the *4690 OS: Messages Guide* for a list of system error codes.

## Write Bitmap Data to the Printer

This function is designed to make printing to the middle 300 dots of a print line easy to specify.

### 16-Bit Syntax

```
long adx_twrite_logo(int session,
                    unsigned char _far *logo);
```

### 32-Bit Syntax

```
long adx_twrite_logo(short session,
                    unsigned char *logo);
```

Refer to the *4680 BASIC: Language Reference* for the format of the print data.

If the operation is successful, the function returns 0. Otherwise, a negative value is returned and the error code and the session number are placed in global variables *adx\_errn* and *adx\_errf*. See the *4690 OS: Messages Guide* for a list of system error codes.

## Flush Pending Writes to the Printer

This function is valid only for the printer. To suspend the application until all queued print lines are printed at all of the print stations, issue an *adx\_ttclose* using the session number of any of the open print stations.

### 16-Bit Syntax

```
int adx_ttclose(int session);
```

## 32-Bit Syntax

```
short adx_ttclose(short session);
```

## Read Data from the Fiscal Printer

This function reads data from the fiscal printer.

## 16-Bit Syntax

```
int adx_tfiscal_read(unsigned char _far *buffer,  
                    int bufsize);
```

## 32-Bit Syntax

```
short adx_tfiscal_read(unsigned char *buffer,  
                      short bufsize);
```

The buffer size must be at least 266 bytes.

If the operation is successful, the function returns 0. Otherwise, a negative value (-1) is returned and the error code is placed in global variable *adx\_errn*. See the *4690 OS: Messages Guide* for a list of system error codes. *adx\_errf* is set to 0.

Use the following sequence to read fiscal printer data:

- Write the FISCAL READ command (0xda or 0xdb) to the printer with *adx\_twrite\_device()*.
- Call *adx\_ttclose()* to flush the print buffer.
- Call *adx\_tfiscal\_read()* to receive fiscal data.

See “Fiscal Printer Support” on page 137 for the format of the data to read from the fiscal printer.

---

## Miscellaneous Routines

This section lists some miscellaneous non-I/O services provided by the CPI. With the exception of the memory allocation functions, there is a BASIC counterpart for each of these C routines. Wherever possible, argument names are the same as or similar to the corresponding BASIC names. In most cases, the descriptions of BASIC functions apply equally to the CPI. Differences are noted where they occur.

As a general rule, a negative return value (-1) from a CPI function indicates that an error has occurred while the function is running. The error code is placed in global variable *adx\_errn*. See the *4690 OS: Messages Guide* for a list of system error codes.

## Allocate Heap Space

**Attention:** This function is valid for 16-bit interfaces only.

This function allocates uninitialized storage from the application heap. If the space cannot be allocated, *adx\_tmalloc()* returns NULL. Otherwise, it returns a pointer to the newly allocated space.

**Note:** Memory allocation routines such as *malloc* and *free*, which are shipped with standard C libraries, should not be called from 4690 terminal applications. Attempting to do so has unpredictable results.

The heap is managed by the CPI for use by both the application and the API. In medium memory model it occupies a portion of the application data segment, which is allocated at load time. In large memory model, the heap occupies one or more data segments, which are requested from the OS as needed.

See “Requesting Operating System Memory” on page 432 and “CPI Extended Memory Management for the 469x POS Terminal” on page 445. for other methods of acquiring memory dynamically.

## Syntax

```
void _far *adx_tmalloc(unsigned long size);
```

### Parameters:

**size**    Size in bytes of the space requested.

**Note:** In the 32-bit interface, the C run-time memory allocation routines (malloc/free) should be used for memory allocations (even on the terminal). The CPI routines are not supported.

## Free Heap Space

**Attention:** This function is valid for 16-bit interfaces only.

This function releases the data item pointed to by *ptr*. Argument *ptr* must be a pointer that was returned by *adx\_tmalloc()*.

## Syntax

```
int adx_tfree(void _far *ptr);
```

If the operation is successful, the return value is 0; otherwise, it is -1.

**Note:** In the 32-bit interface, the C run-time memory allocation routines (malloc/free) should be used for memory allocations (even on the terminal). The CPI routines are not supported.

## Query Available Heap Space

**Attention:** This function is valid for 16-bit interfaces only.

This function returns a 4-byte integer indicating the number of free bytes available in the heap.

## Syntax

```
long adx_ttfre(void);
```

## Query Contiguous Available Heap Space

**Attention:** This function is valid for 16-bit interfaces only.

This function returns a 4-byte integer equal to the largest number of contiguous bytes of available memory space in the heap.

## Syntax

```
long adx_tmfre(void);
```

## Chain to Another Program

### 16-Bit Syntax

```
int adx_tchain(char _far *name,  
               CHAINVAR_t _far *arglist,  
               unsigned int nargs);
```

where:

```
typedef struct chainvar {  
    unsigned int size;    // argument size in bytes  
    void _far *arg;      // address of the argument  
} CHAINVAR_t;
```

## 32-Bit Syntax

```
short adx_tchain(char *name,
                 CHAINVAR_t *arglist,
                 unsigned short nargs);

where:
    typedef struct chainvar {
        unsigned short size;    // argument size in bytes
        void *arg;             // address of the argument
    } CHAINVAR_t;
```

### Parameters:

**name** Name of the program to be loaded and executed. The maximum name length is 26 bytes, including the null terminator.

**arglist** Array of arguments to be passed to the new program.

**nargs** Number of arguments in the argument list.

See the *4680 BASIC: Language Reference* for a complete description of the chaining operation.

## Usage

The arguments are written to a pipe by the chaining program. The chained-to program calls *adx\_tuse()* to read the arguments from the pipe. Because they are read from the pipe in the same order in which they were written, the arglists passed to *adx\_tchain()* and *adx\_tuse()* must exactly match in number, order and size of arguments in order for the arguments to be passed successfully between programs.

If the chaining operation is unsuccessful, *adx\_tchain()* returns a negative value (-1) and a return code is placed in global variable *adx\_errn*. See the *4690 OS: Messages Guide* for a list of system error codes.

## Receive Parameters From Chaining Program

This function reads data that was written to a pipe by the application that initiated execution of the current application using *adx\_tchain()*.

## 16-Bit Syntax

```
int adx_tuse(CHAINVAR_t _far* arglist,unsigned int nargs);
```

```
where:
    typedef struct chainvar {
        unsigned int size;    // argument size in bytes
        void _far *arg;       // address of the argument
    } CHAINVAR_t;
```

## 32-Bit Syntax

```
short adx_tuse(CHAINVAR_t * arglist,unsigned short nargs);
```

```
where:
    typedef struct chainvar {
        unsigned short size;  // argument size in bytes
        void *arg;           // address of the argument
    } CHAINVAR_t;
```

### Parameters:

**arglist** Array of arguments to be passed to the new program.

**nargs** Number of arguments in the argument list.

The caller must provide sufficient buffer space for each argument. Because the CPI does not know the data type of the arguments it copies into the caller's data space, it cannot null-terminate strings. The caller must append nulls as required, perhaps by filling string buffers with nulls before calling *adx\_tuse()*.

The function returns the smaller of:

1. Amount of data in the pipe
2. Amount of data requested by the application

## Get System Date

The function writes a string in the format *YYYYMMDD* in the caller's buffer. *YYYY* is the year and ranges from 0000 to 9999; *MM* is the month and ranges from 01 to 12; *DD* is the day and ranges from 01 to 31. The caller's buffer must be at least 9 bytes long to allow for a null terminator.

### 16-Bit Syntax

```
long adx_tdate(char _far *buffer);
```

### 32-Bit Syntax

```
long adx_tdate(char *buffer);
```

ADX\_TDATE returns 0 for successful completion, or 1 if the buffer argument is a null pointer or if an operating system error has occurred.

## Get System Time

The function writes a string in the format *HHMMSS* in the caller's buffer. *HH* is the hour of the day and ranges from 0 to 24; *MM* is the minutes in an hour and *SS* is the seconds in the minute. Both of these range from 1 to 60. The caller's buffer must be at least 7 bytes long to allow for a null terminator.

### 16-Bit Syntax

```
int adx_ttime(char _far *buff);
```

### 32-Bit Syntax

```
short adx_ttime(char *buff);
```

## Disable Software Interrupts

### 16-Bit Syntax

```
int adx_tdisable(void);
```

### 32-Bit Syntax

```
short adx_tdisable(void);
```

This function returns a 0 if successful; -1 if not.

## Enable Software Interrupts

### 16-Bit Syntax

```
int adx_tenable(void);
```

### 32-Bit Syntax

```
short adx_tenable(void);
```

This function returns a 0 if successful; -1 if not.

## Pack Data

This function operates on strings. It strips off the four most significant bits of each character, then packs the resulting low-order 4 bits, two per byte.

## 16-Bit Syntax

```
int adx_tpack(unsigned char _far *packed,  
              unsigned char _far *unpacked,  
              unsigned int length);
```

## 32-Bit Syntax

```
short adx_tpack(unsigned char *packed,  
                unsigned char *unpacked,  
                unsigned short length);
```

This function returns a 0 if successful; -1 if not. For more information, see the *PACK\$* function in the *4680 BASIC: Language Reference*.

### Parameters:

#### **packed**

Output buffer for the packed string. The application must ensure that the buffer is large enough to contain the packed string plus a null terminator.

#### **unpacked**

String to be packed.

**length** Length of the string to be packed.

## Unpack Data

This function operates on strings. It reverses the action of the *adx\_tpack()* function.

## 16-Bit Syntax

```
int adx_tunpack(unsigned char _far *unpacked,  
                unsigned char _far *packed,  
                unsigned int length);
```

## 32-Bit Syntax

```
short adx_tunpack(unsigned char *unpacked,  
                  unsigned char *packed,  
                  unsigned short length);
```

This function returns a 0 if successful; -1 if not. For more information, refer to the *UNPACK\$* function in the *4680 BASIC Language Reference*.

### Parameters:

#### **unpacked**

Output buffer for the unpacked string. The application must ensure that the buffer is large enough to contain the unpacked string plus a null terminator.

#### **packed**

Input string to be unpacked.

**length** Length of the string to be unpacked.

## Requesting Operating System Memory

**Attention:** This function is valid for 16-bit interfaces only.

This function requests an allocation of additional memory from the operating system. This function should be used primarily to allocate large, relatively permanent buffers. Use *adx\_tmalloc* to allocate memory from the CPI heap.

**Note:** The application must use 32-bit pointers to reference memory acquired by calling *adx\_tmem\_get*.

## Syntax

```
int adx_tmem_get(unsigned int option,  
                unsigned long _far *mpbptr);
```

### Parameters:

#### option

0 = Expand existing segment

1 = Allocate a new segment

#### mpbptr

Address of memory parameter block

0 = start

4 = min

8 = max

where:

**start** For option = 0, set the base address of the segment to be expanded in this field. The base address of the added memory portion is put here when the allocation has completed. If the segment has already reached its 64-KB limit, a new segment can be allocated.

For option = 1, set this field to 0. The base address of the new segment is put here.

**min** Specify the minimum number of bytes required. The actual number of bytes allocated is returned in this field.

**Note:** For all usage (either option 1 or 0) the amount of memory allocated can be less than the value set up in the *min* field of the parameter block. The call fails only if no more memory can be allocated. Therefore it is imperative that you always check the value of *min* on the function's return.

**max** Specify the maximum number of bytes required. This value is not changed.

This function either adds contiguous memory to the end of an existing segment or allocates a new segment. Use the option field to select one or the other and the Memory Parameter Block to specify the minimum and maximum memory requirements. Set the Memory Parameter Block's start parameter to the base address of an existing data segment for option 0 or to zero for option 1. Option 1 must be specified the first time this function is called.

**Note:** Medium Memory Model users should not get a full 64K segment of data due to the dangers of undetected rollover when manipulating data near the end of the segment. Specify a maximum value no greater than 65,532 bytes.

When option 0 is selected, the designated segment is extended contiguously. The Memory Parameter Block minimum and maximum allocation fields (*min* and *max*) are the amounts by which the heap is to grow, and not the new total size of the heap.

The call can also result in a new memory segment being allocated. Therefore, the programmer should check the return value in *start* to get the starting address of the new memory. The Memory Parameter Block start address (*start*) and minimum allocation (*min*) are modified to reflect the new starting address and actual allocation, respectively.

When option 1 is selected, newly allocated memory might or might not be contiguous with any previously allocated memory. This function modifies the Memory Parameter Block's *start* and *min* values to indicate the new base address and actual allocation. It is possible for the new segment to be allocated such that an existing segment is no longer expandable.

If the operation is successful, the function returns 0. If the operation fails, the function returns -1 and places an error code in the global variable **adx\_errn**. See the *4690 OS: Messages Guide* for a list of system error codes.

## Freeing Storage

**Attention:** This function is valid for 16-bit interfaces only.

This function releases memory that has been acquired by a call to *adx\_tmem\_get* from the address specified to the end of the segment.

### Syntax

```
int adx_tmem_free(void _far *mstart);
```

#### Parameters:

##### mstart

Starting address of memory to free.

If the operation is successful, the function returns 0. If it fails, it returns -1 and places an error code in global variable *adx\_errn*. See the *4690 OS: Messages Guide* for a list of system error codes.

## Suspending Processing for Indicated Duration

This function delays the calling process until the specified time or until the specified period of time expires.

### 16-Bit Syntax

```
int adx_ttimer_set(unsigned int flags,  
                  unsigned long stime);
```

### 32-Bit Syntax

```
short adx_ttimer_set(unsigned short flags,  
                    unsigned long stime);
```

#### Parameters:

##### flags

1 = Absolute

0 = Relative

**stime** If flags = 1, stime is the number of milliseconds to delay after midnight. If flags = 0, it is the number of milliseconds to delay.

If absolute time is specified and the current time of day is beyond it, the process delays until the specified time on the next day.

If the operation is successful, the function returns 0. If it fails, it returns -1 and places an error code in global variable *adx\_errn*. See the *4690 OS: Messages Guide* for a list of system error codes.

## Waiting for Data

The application uses *adx\_twait()* to wait for data to become available from one or more input devices or pipes. It suspends program processing while the status of the specified devices or pipes is monitored. Execution resumes when data becomes available or the timeout interval is exhausted, whichever occurs first. The first wait to be satisfied is indicated by the function return value.



## 16-Bit Syntax

```
int adx_twait (unsigned long wtime,  
              unsigned int wcount,  
              int _far *sessions);
```

## 32-Bit Syntax

```
short adx_twait (unsigned long wtime,  
                unsigned short wcount,  
                short *sessions);
```

### Parameters:

**wtime** Time to wait, in milliseconds. For example, to wait 1 second set `wtime = 1000`. If a time limit of 0 is specified, the operating system waits indefinitely until at least one of the waits is satisfied.

### wcount

Number of pipes and devices to wait for. This count should be the actual count of wait events to initiate and not the maximum. For most applications, this count is less than 10 and usually 1 or 2. Maximum count = 30.

### sessions

Array of the session numbers of the open devices or pipes. The size of this array is specified by parameter *wcount*.

Return value:

- If the return value is 0, the specified time expired before an event completed.
- A positive return value is the position in the session number array of the first device or pipe to have data available. For example, if there are three session numbers in the array and the wait completes for the third one, the return value is 3.  
If multiple WAITs are satisfied simultaneously, the one that is first in the array is returned.
- A negative return value (-1) indicates that an error has occurred and an error code placed in global variable *adx\_errn*. See the *4690 OS: Messages Guide* for a list of system error codes.

For pipes, the wait for data to be available in a pipe is satisfied when at least one byte is in the pipe.

The maximum number of waits that can be requested at a time is 30 minus the number of other asynchronous events outstanding, such as uncompleted print lines to the printer, and uncompleted writes to the serial I/O driver and coin dispenser.

## Converting Hex to ASCII

This function converts four hexadecimal bytes to eight bytes of ASCII characters. This can be used to convert the system error codes from hexadecimal characters to printable ASCII characters.

## 16-Bit Syntax

```
char _far *adx_tprthex(char _far *anum,  
                      unsigned long hnum);
```

## 32-Bit Syntax

```
char *adx_tprthex(char *anum,  
                 unsigned long hnum);
```

### Parameters:

**anum** Buffer to receive ASCII characters. It must be at least 9 bytes in length to permit a null terminator to be appended.

**hnum** Hexadecimal number to convert.

The function returns a pointer to the converted ASCII string or a NULL if an error occurs.

---

## Terminal Application Services

The CPI Terminal Application services provide a variety of functions for terminal applications written in C language. This interface is similar to the 4680 BASIC Terminal Applications Services interface described in Chapter 16, “Designing applications with 4680 BASIC,” on page 277. Differences between the functions are noted where they occur.

### ADX\_TSERVE (Requesting an Application Service)

The function code, parameters, and explanation for each service available under the *adx\_tserve* function are shown below.

These parameters are slightly different from those described for ADXSERVE in Chapter 16, “Designing applications with 4680 BASIC,” on page 277.

#### 16-Bit Syntax

```
long adx_tserve (unsigned int funcnum,  
                char _far *databuff,  
                unsigned int dbuffsize);
```

#### 32-Bit Syntax

```
long adx_tserve (unsigned short funcnum,  
                char *databuff,  
                unsigned short dbuffsize);
```

#### Parameters:

##### **funcnum**

Function number for the service requested.

##### **databuff**

Buffer for data sent to service requested or for data received from service requested.

##### **dbuffsize**

Size (bytes) of databuff or data for requested service if databuff is not used.

A 0 is returned if the operation is successful; a -1 if not. See the *4690 OS: Messages Guide* for a list of system error codes.

### Dump System Storage

The Dump System Storage function causes all of the storage in the terminal at which the request is made to be dumped to disk file *adxcsltf.dat* on the controller. Both the application and operating system storage are dumped.

The function uses the following parameters:

##### **funcnum**

= 1

##### **databuff**

= Unused

##### **dbuffsize**

= Unused

## Enable Terminal Storage Retention

This function enables the terminal's storage retention. The effect is temporary. The condition established in the controller overrides the temporary terminal condition. Whenever the terminals receive updates from the controller, terminal online updates occur when terminal-to-controller communications are interrupted or when system functions are used that require sending new status to a terminal.

The function uses the following parameters:

**funcnum**  
= 2

**databuff**  
= Unused

**dbuffsize**  
= Unused

## Disable Terminal Storage Retention

This function disables the terminal's storage retention.

The function uses the following parameters:

**funcnum**  
= 3

**databuff**  
= Unused

**dbuffsize**  
= Unused

## Get Application Status Information

This function returns the terminal's status information and places it in **databuff**. The information is in ASCII format. The data for the terminal application status is:

The function uses the following parameters:

**funcnum**  
= 4

**databuff**  
= Address of buffer to receive status information.

**dbuffsize**  
= 80 bytes

Offset	Length	Description
0	4	Store number
4	1	Date format 1 = m/d/y 2 = d/m/y 3 = m.d.y 4 = d.m.y
5	1	Time format 1 = h:m:s 2 = h.m.s

Offset	Length	Description
6	1	Monetary format 1 = period convention (1,234,970.06) 2 = comma convention (1.234.970,06)
7	3	Terminal number 001–999
10	1	Terminal online/offline 0 = offline 1 = online
11	1	Storage Retention 0 = disabled 1 = enabled
12	24	Default Application name • device name — 8 character maximum (:) • file name — 8 character maximum (.) • file extension — 3 character maximum
36	1	Number of digits after decimal point 0 or 2
37	1	Terminal powered on/off 0 = on 1 = off — Always on for Mod1; can be on or off for Mod2.
38	1	Not applicable
39	1	Backup (loop) 1 = Loop in backup 0 = Not in backup
40	1	System Display 1 = Display named ANDISPLAY is the system display 2 = Display named ANDISPLAY2 is the system display 3 = Display named VDISPLAY is the system display 4 = Display named VDISPLAY2 is the system display 5 = Display named ANDISPLAY3 is the system display
41	1	Terminal type 0 = Mod1 1 = Mod2
42	3	Partner terminal address
45	2	Current store controller ID
47	1	Video display adapter 0 = 4683 Video display feature A 1 = VGA
48	1	Application Environment 0 = Running in a terminal 1 = Running in a controller/terminal
49	2	Hard Totals NVRAM size 01 = 1K bytes (4683) 16 = 16K bytes (4693)

Offset	Length	Description
51	1	Terminal Type <ul style="list-style-type: none"> <li>• 1 (4693)</li> <li>• 2 (4694)</li> <li>• 3 (4683)</li> <li>• 4 (4684)</li> <li>• 5 (SurePOS 300/700 Series, TCxWave 6140 Series)</li> </ul>
52	1	Supporting Operating System <ul style="list-style-type: none"> <li>• 1 (4690 OS)</li> <li>• 2 (4690 Terminal Services for DOS)</li> </ul>
53	1	Printer Device Type <p>X'00' — Pre-4610 printer attached</p> <p>X'30' — 4610 printer attached</p>
<b>Note:</b> If Device type is X'00' offset 54 through offset 57 are also X'00'.		
54	1	Printer Device ID <p>X'00' — Model TI1 or TI2 (Impact DI/Thermal CR)</p>
55	1	Printer Features <p>Bit 0 (X'01') set to 0 — No MICR is present</p> <p>Bit 0 (X'01') set to 1 — MICR is present</p> <p>Bit 1 (X'02') set to 0 — No check flipper is present</p> <p>Bit 1 (X'02') set to 1 — Check flipper is present</p> <p>Bit 2 (X'04') set to 0 — SBCS (single-byte character set) printer</p> <p>Bit 2 (X'04') set to 1 — DBCS (double-byte character set) printer</p>
56	1	Printer Command Set in use <p>X'00' — Command Set version 1</p>
57	1	EC level of code loaded in the printer
58	1	Java redirection information <p>Bit 7 (0x80) set to 1 — I/O Processor redirected</p> <p>Bit 6 (0x40) set to 1 — ANDISPLAY1 handler installed</p> <p>Bit 5 (0x20) set to 1 — ANDISPLAY2 handler installed</p> <p>Bit 4 (0x10) set to 1 — Cash Receipt Station monitor installed</p> <p>Bit 3 (0x08) set to 1 — Cash Receipt Station handler installed</p> <p>Bit 2 (0x04) set to 1 — Document Insert Station handler installed</p> <p>Bit 1 (0x02) set to 1 — Summary Journal Station handler installed</p> <p>Bit 0 (0x01) set to 1 — Magnetic Stripe Reader handler installed</p>

## Programmable power

The programmable power function is used to power off the terminal. The day, hour and minute in the *DDHHMM* string is the power-on time. When Storage retention is enabled, the individual terminal hardware configuration determines if the terminal will power off or go into a standby/suspend state.

### funcnum

= 5

### databuff

= 9-byte buffer containing:

2-byte integer set to 5 (power off local machine)

6-byte null-terminated character string *DDHHMM* (day-hour-minute), where

DD = The day of the month (0–31)

HH = The hours of the day (0–24)

MM = The minutes of the hour (00–59)

**dbuffsize**  
= 9

## **Enable/Disable IPL**

Terminals are able to run offline from the controller. The primary reason to run offline is if there is a problem with the controller.

When the terminals run offline, the application can write the transaction data to the terminal RAM disk. This enables the terminals to run offline while a controller problem is being addressed. The terminals eventually communicate with the controller and transfer all saved transactions in terminal RAM disk to normal transaction files in the controller. If the terminal is IPLed before the saved transactions are moved to the controller, the saved transactions are lost.

Enable/Disable IPL allows an application program that is running in a terminal to temporarily prevent automatic reloading of the terminal. Automatic terminal reload can occur when the terminal is offline and returns online or when an operator requests the Load Terminal Storage Function with an asterisk (\*) specified as the terminal number. Automatic terminal reloads can also occur when ADXCS20L is invoked with the Load All Terminals function code (for controller applications.)

Applications should use the Disable IPL function when the terminal application recognizes that it is offline and begins to save data on the RAM disk. When the terminal application recognizes it is online with the controller, the saved data can be transferred to the controller. After all data has been transferred to the controller, the terminal application can use the Enable IPL function to allow an IPL.

The requests to enable and disable the IPL of the terminal for the 4683-xx1 and the 4683-xx2 are handled independently. If a Disable Terminal IPL was outstanding on both terminals and one terminal's application issued a request to enable a terminal IPL, an IPL is not possible for the 4683-xx1 and the 4683-xx2 pair. Both terminals must have the IPL enabled before an IPL can occur.

## **Enable IPL**

This function enables the terminal to IPL. If the terminal was IPLed earlier, but could not because the user had disabled the IPL, requesting this function would cause the terminal to IPL.

This function uses the following parameters:

**funcnum**  
= 53

**databuff**  
= Unused

**dbuffsize**  
= Unused

## **Disable IPL**

This function prevents the automatic reload that might occur when a terminal comes online. This function enables the terminal application to effectively use the terminal RAM disk support for temporarily logging data. In most cases, when the controller returns online, the terminal application transfers the saved transaction files to the controller.

The function uses the following parameters:

**funcnum**  
= 54

**databuff**  
= Unused

**dbuffsize**  
= Unused

## Switching between the terminal Java, terminal application, and enhanced mode graphical extensions screens

On a controller/terminal, the system must be on the terminal side for these APIs to display either the terminal Java screen (Java 2 or Java 6, the terminal application screen, or the enhanced mode graphical extensions screen. If a controller screen or command line is being displayed and a terminal application calls this API, you will not be switched away from a controller screen.\

This function is not available for 16bit C.

### Switch to Java screen

This function uses the following variables:

**funcnum**  
= 55

**databuff**  
= 0

**dbuffsize**  
= 1

This function is used to switch the terminal video so that the terminal Java screen is displayed. It has the same effect as typing **Alt+SysRq+J** on an ANPOS or standard keyboard, or typing S1,5,S2 on the POS keyboard when the terminal application screen is displayed.

### Switch to terminal screen

This function uses the following variables:

**funcnum**  
= 55

**databuff**  
= 0

**dbuffsize**  
= 2

This function is used to switch the terminal video so that the terminal Java screen is displayed. It has the same effect as typing **Alt+SysRq+T** on an ANPOS or standard keyboard, or typing S1,5,S2 on the POS keyboard when the terminal application screen is displayed.

### Switch to enhanced mode graphical extensions screen

This function uses the following variables:

**funcnum**  
= 55

**databuff**  
= 0

**dbuffsize**  
= 4

This function is used to switch the terminal video so that the enhanced mode graphical extensions screen is displayed. It has the same effect as typing **Alt+SysRq+X** on an ANPOS or standard keyboard, or typing S1,58,S2 on the POS keyboard. This function is not available on Classic systems.

## ADX\_TERROR (Log an Application Event)

This function can be called from a terminal application to log an error in the terminal error log and optionally display system messages.

### 16-Bit Syntax

```
int adx_terror (unsigned int term,
               unsigned int      msggrp,
               unsigned int      msgnum,
               unsigned int      severity,
               unsigned int      event,
               unsigned char *_far * unique,
               unsigned int      ulen);
```

### 32-Bit Syntax

```
short adx_terror (unsigned short term,
                 unsigned short      msggrp,
                 unsigned short      msgnum,
                 unsigned short      severity,
                 unsigned short      event,
                 unsigned char *      unique,
                 unsigned short      ulen);
```

A return value of 0 signifies a successful copy. A negative value is returned if an error occurs. See “CPI Error Codes” on page 450 for a list of error codes.

#### Parameters:

**term** Terminal number. This information can be obtained from the Get Application Status Information function.

#### msggrp

Two-byte integer that contains the ASCII equivalent of the message group character used to identify the error messages for a product. This character is unique for each product and can be in the range J to S. For example, **msggrp** = 75 for the letter K.

#### msgnum

Two-byte integer message number, if any. If the message number is zero, no message are displayed. This number is converted to three printable decimal digits and appended to the message group letter to form the message identifier. This identifier is used to search the Application Message file for a message to display.

#### severity

Two-byte integer ranging from 1 to 5 that is used to indicate the importance of each message. This number is a message level indicator with the most important messages as severity = 1. An operator can request messages to be displayed. The messages with severity level less than or equal to the level requested are displayed.

**event** Two-byte integer that can be set by the application to indicate a specific situation or problem. This must be in the range 64 to 79 for the controller. The system uses the log data to generate Network Problem Determination Alert messages.

#### unique

Short string of characters to be included in message. The maximum length is 10 bytes. If the message is longer than 10 bytes, only the first 10 bytes are used. If the message is less than 10 bytes, blanks are added.

**ulen** Number of characters in unique message including imbedded blanks.



## ADX\_TDIR (Listing Terminal RAM/Hard Disk Files)

This function is used to list the files in RAM disks X: and Y: and files on the terminal hard disk C:. It also lists the amount of free space on the disks. The function is typically called within a loop in an application, such that each repetition of the loop passes the information for a directory entry. The *opt* parameter should indicate the first entry on the first repetition of the loop, and it should indicate the next entry on the following repetition. When the application is executing such a loop, the function passes information for each file on the RAM or hard disk in the *direntry* string as the loop progresses. As the loop is executed and file information is passed in **direntry**, the application can display it, collect it, or send it to an application in the controller, for example. When information for all existing files has been retrieved, the function return value indicates this, and the application can end the loop.

### 16-Bit Syntax

```
typedef struct {
    char      filename[14] // null-terminated
    unsigned long filesize;
    char      date[8]
    char      time[6]
    unsigned int numfiles;
    unsigned long freebytes; // free space on the disk
} DIRDATA_t;

int adx_tdir(char _far *disk,
             DIRDATA_t _far *direntry,
             unsigned int opt);
```

### 32-Bit Syntax

```
typedef struct {
    char      filename[14] // null-terminated
    unsigned long filesize;
    char      date[8]
    char      time[6]
    unsigned short numfiles;
    unsigned long freebytes; // free space on the disk
} DIRDATA_t;

short adx_tdir(char *disk,
               DIRDATA_t *direntry,
               unsigned short opt);
```

### Parameters:

**disk** String containing the disk id (X:, Y: or C:) and optionally continuing with a file name with or without wildcards (\*).

### direntry

String that contains information for one directory entry on return to the caller.

**opt** Option indicating whether the first or next directory entry is wanted. An integer value of 0 indicates first and 1 indicates next.

The following return code values indicate no error:

- 0 - successful passing of file information
- 1 - no more files on the disk

If the function returns 0, all the fields in *direntry* are filled in. If the return value is 1, the *numfiles* and *freebytes* fields are filled in, the string fields are null, and *filesize* is zero.

A negative return value (-1) indicates that an error has occurred. See “CPI Error Codes” on page 450 for a list of error codes.

## ADX\_TCOPYF (Copying Disk Files)

This function is used for copying disk files. It is designed specifically for RAM and hard disk files, but can be used on any files.

### 16-Bit Syntax

```
int adx_tcopyf(char _far *infile,
               char _far *outfile,
               unsigned int opt0,
               unsigned int opt1);
```

### 32-Bit Syntax

```
short adx_tcopyf(char *infile,
                 char *outfile,
                 unsigned short opt0,
                 unsigned short opt1);
```

#### Parameters:

**infile** String containing file name to be copied from. The total length of a controller input file name must be less than 25 characters. You can use a logical name to avoid this restriction.

**outfile** String containing the file name to be copied to.

**opt0** 2-byte integer indicating whether copy operation should be performed.

**0** File is copied regardless of whether the output file already exists.

**1** Error is returned if the file already exists and the file is not copied.

**opt1** 2-byte integer used to indicate distribution on a LAN system.

**0** Keep the distribution on the output file the same as on the input file.

**1** If the output file exists, keep the same distribution.

**2** Make the output file local.

A return value of 0 signifies a successful copy. A negative value (-1) is returned if an error occurs. See “CPI Error Codes” on page 450 for a list of error codes.

**Note:** This function is applicable for copying files from the controller to the terminal.

## ADX\_TCRYPT (Encrypting a Password)

This function encrypts an eight-character ASCII password. The operating system uses one-way encrypted passwords for authorization to sign on to the system. The encryption method can also be used by an application to establish authorization for use of applications, data files, or other things that need access controlled by the application.

The password to be encrypted should be an ASCII string of up to 8 alphanumeric characters with no leading blanks. Trailing blanks are ignored. Leading zeroes are counted as part of the password. The encrypted value returned in *pwout* is an eight-character string of ASCII characters that represent decimal digits.

### 16-Bit Syntax

```
int adx_tcrypt(char _far *pwin, char _far *pwout);
```

### 32-Bit Syntax

```
short adx_tcrypt(char *pwin, char *pwout);
```

**Parameters:**

**pw**in Password to be encrypted. Must be terminated with a NULL if less than eight characters.

**pw**out Encrypted value returned.

The function returns 0 if the operation is successful. A negative return value indicates an error. Error codes are listed in “CPI Error Codes” on page 450.

---

## CPI Extended Memory Management for the 469x POS Terminal

**Attention:** Extended memory management is not supported by the 32-bit interface.

This section includes Terminal Extended Memory Management routines. This interface is similar to the 4680 BASIC Extended Memory Management interface described in Chapter 16, “Designing applications with 4680 BASIC,” on page 277. In most cases, descriptions of functions and their arguments apply equally to the C API. Differences are noted where they occur.

### Error Handling

The Extended Memory Management routines in this section do not have the *retcode* and *sysret* parameters that are defined for BASIC routines. Instead, they indicate failure by returning a negative error code. If successful, they return one of the following codes:

0, or  
> 0 (*adx\_tgetmem* or *adx\_topenmem* only)

If unsuccessful, they return one of the following values:

A negative value listed at the end of this chapter, or  
-1

In addition, in the event of an error, an error code, if one is available, is placed in *adx\_errn*. A file ID is also placed in *adx\_errf*, if available. The error codes that are placed in *adx\_errn* are listed in “CPI Error Codes” on page 450.

**Note:** Error code 1019 (“Receiving string not large enough”) is not returned by the CPI because this is an error condition that it cannot detect.

See the *4690 OS: Messages Guide* for a list of system error codes.

### Data Buffers

BASIC routines use the BASIC string format to check the size of the caller's data buffer and return an error if it is too small for the intended operation. C cannot do this. The application should allocate a buffer at least as large as the length parameter specified on the function call. Failure to do so can introduce problems that are extremely difficult to diagnose.

### Allocating a Memory File

#### Syntax

```
long adx_tgetmem(unsigned int fileid,  
                unsigned long kount,  
                unsigned int recsize,  
                unsigned int shflag);
```

**Parameters:**

**fileid** Memory file number that is used for subsequent accesses.

**kount** Number of memory records to allocate.

**recsize**

Memory file record size.

**shflag** Shared memory flag.

0 = not shared

1 = shared

**Return values:**  $\geq 0$  if successful; this represents a record count. If an error occurred, one of the negative error codes listed in “Error Codes” on page 449 is returned.

## Accessing a Shared Memory File

### Syntax

```
long adx_topenmem(unsigned int fileid,  
                  unsigned int recsize);
```

**Parameters:**

**fileid** Memory file number that was specified in the call to *adx\_tgetmem*.

**recsize**

Memory file record size.

**Return values:**  $\geq 0$  if successful; this represents a record count. If an error occurs, one of the negative error codes listed in “Error Codes” on page 449 is returned.

## Gaining Mutually Exclusive Access to Shared Memory

### Syntax

```
long adx_tmemsyn(unsigned int fileid);
```

**Parameters:**

**fileid** Memory file number that was specified on the call to *adx\_tgetmem*.

**Return values:** 0 if successful. If an error occurs, one of the negative return codes listed in “Error Codes” on page 449 is returned.

## Releasing Exclusive Access to Shared Memory

### Syntax

```
long adx_tmemunsyn(unsigned int fileid);
```

**Parameters:**

**fileid** Memory file number that was specified on the call to *adx\_tgetmem*.

**Return values:** 0 if successful. If an error occurs, one of the negative return codes listed in “Error Codes” on page 449 is returned.

## Freeing a Memory File

### Syntax

```
long adx_tfreemem(unsigned int fileid);
```

**Parameters:**

**fileid** Memory file number that was specified on the call to *adx\_tgetmem*.

**Return values:** 0 if successful. If an error occurs, one of the negative return codes listed in “Error Codes” on page 449 is returned.

## Querying Available Free Memory

### Syntax

```
long adx_tavailmem(void);
```

## Writing to a Memory File

### Syntax

```
long adx_tmemwrite(unsigned int fileid,  
                   unsigned char _far *data,  
                   unsigned long recnum,  
                   unsigned int offset,  
                   unsigned int length);
```

#### Parameters:

**fileid** Memory file number that was specified on the call to *adx\_tgetmem*.

**data** Address of data buffer.

**recnum**  
Record number.

**offset**  
Offset into the record.

**length**  
Length of data (0 does not default to the record size).

**Return values:** 0 if successful. If an error occurs, one of the negative error codes listed in “Error Codes” on page 449 is returned.

## Reading From a Memory File

### Syntax

```
long adx_tmemread(unsigned int fileid,  
                  unsigned char _far *data,  
                  unsigned long recnum,  
                  unsigned int offset,  
                  unsigned int length);
```

#### Parameters:

**fileid** Memory file number that was specified on the call to *adx\_tgetmem*.

**data** Address of data buffer.

**recnum**  
Record number.

**offset**  
Offset into the record.

**length**  
Length of data (0 does not default to the record size).

**Note:** The application must ensure that the data buffer is large enough to receive the data.

Return values: 0 if successful. If an error occurs, one of the error codes listed in “Error Codes” on page 449 is returned.

## Binary Search for Matching Field

### Syntax

```
long adx_tmemsrchb(unsigned int fileid,  
                  unsigned char _far *data,  
                  unsigned int slength,  
                  unsigned int toffset,  
                  unsigned int rlength,  
                  unsigned long roffset);
```

#### Parameters:

##### Description

**fileid** Memory file number that was specified on the call to *adx\_tgetmem*.

**data** Address of data buffer.

This buffer holds the search string prior to the function call. If a matching record is found, the data read from the file is placed in this buffer. The caller must ensure that the buffer is large enough to receive the data.

**slength**

Length of the search string.

**toffset**

Offset into the record of the field to be searched.

**rlength**

Length of return data. If 0, no data is returned.

**roffset**

Offset of return data field in record.

**Return values:** 0 if successful. If the record is not found, or an error has occurred, one of the negative return codes listed in “Error Codes” on page 449 is returned.

## Sequential Search for Matching Field

### Syntax

```
long adx_tmemsrchs(unsigned int fileid,  
                  unsigned char _far *data,  
                  unsigned int slength,  
                  unsigned int toffset,  
                  unsigned int rlength,  
                  unsigned long roffset);
```

See the description of *adx\_tmemsrchb* for a definition of the parameters and return values for this function.

## Writing Data to a Memory File in Ascending Order by Key

### Syntax

```
long adx_tmemwrkey(unsigned int fileid,  
                  unsigned char _far *data,  
                  unsigned int klength,  
                  unsigned int length);
```

#### Parameters:

### Description

**fileid** Memory file number that was specified on the call to *adx\_tgetmem*.

**data** Address of data buffer.

**klength**  
Length of key.

**length**  
Length of data.

**Return values:** If data was written, the number of bytes written is returned. If an error occurs, one of the negative return codes listed in “Error Codes” on page 449 is returned.

## Deleting Data from a Memory File Sequenced by Key

### Syntax

```
long adx_tmmdlkey(unsigned int fileid,  
                 unsigned char_far *key,  
                 unsigned int klength);
```

### Parameters:

**fileid** Memory file number that was specified on the call to *adx\_tgetmem*.

**key** Key of the record to be deleted.

**klength**  
Length of the key.

**Return values:** 0 if successful. If an error occurs, one of the negative return codes listed in “Error Codes” on page 449 is returned.

## Clearing a Memory File

### Syntax

```
long adx_tmclear(unsigned int fileid);
```

### Parameters:

**fileid** Memory file number that was specified on the call to *adx\_tgetmem*.

**Return values:** 0 if successful. If an error occurs, one of the negative return codes listed in “Error Codes” on page 449 is returned.

## Error Codes

The following error codes are generated by the Extended Memory Management routines:

- 1009 Attempted to allocate more than 64 files.
- 1011 Out of memory; cannot continue.
- 1013 Attempted to allocate the same file ID twice.
- 1014 File not found, or attempted to use *adx\_tmemsyn* or *adx\_tmemunsyn* on a non-shared file.
- 1015 Record position was outside the file.
- 1016 Search record was not found.
- 1017 Invalid data length (greater than 512 bytes).

**-1020** Requested shared memory greater than 65524 bytes in length.

See the *4690 OS: Messages Guide* for a list of system error codes.

---

## Product Signature

It is customary to embed a signature in the run-time libraries that becomes part of the code or data of any application linked with the libraries. This signature is visible in a binary file or in a memory dump. Its purpose is to aid in diagnosing problems, and it contains the library name, last change date, and latest apar number. The format of the signature in the CPI libraries is:

```
ADXCAPITM APAR=IRxxxxx mm/dd/yy (CAPITMED.L86)
or
ADXCAPITL APAR=IRxxxxx mm/dd/yy (CAPITLRG.L86)
```

**Note:** There is no product signature in the 32-bit version of the CPI library (capit.lib) because this is only an import library. The code that implements the CPI functionality is in CAPIT.DLL that is shipped with the 4690 OS and is maintained through normal product maintenance.

---

## Validation of Application Buffers

The CPI validates each buffer passed to it by the application before attempting to read or write to that buffer. To validate a buffer, the CPI determines whether the segment denoted by the buffer's address is reachable from the current privilege level and whether the indicated operation (read or write) is permitted. Then, it verifies that the buffer offset lies within the segment. If the buffer size can be determined, the CPI checks that the entire buffer fits within the segment.

**Note:** Buffer validation is not done by the 32-bit interface code. If an application passes an invalid pointer to the API, a page fault exception is generated and the application is terminated (or a dump is performed if the dump flag is on).

---

## CPI Error Codes

When an error occurs during processing of a supervisor call, the operating system returns a negative error code that is placed in *adx\_errn*. See the *4690 OS: Messages Guide* for a list of system error codes.

When the CPI traps an error it places a positive error code in the global variable *adx\_errn*. These CPI error codes are listed below. If the failing operation involves a file, device or pipe, the associated session number is placed in global variable *adx\_errf*. The application can reference these two global variables to determine the cause of the error.

## General Errors

0x00000101	ERR_GE_FNUM_TABLE_FULL	An open or create attempt failed because there are no more session numbers available.
0x00000102	ERR_GE_FUNC_NOT_VALID	The function requested is not valid for this device.
0x00000103	ERR_GE_INVALID_SESSNUM	An invalid session number was specified.
0x00000104	ERR_GE_BAD_PRIVILEGE	An invalid buffer address was specified; the selector is not at the correct privilege level.
0x00000105	ERR_GE_BAD_ACCESS	An invalid buffer address was specified; an invalid access was attempted, (for example, writing into a code segment).



0x00000106	ERR_GE_BAD_OFFSET	An invalid buffer address was specified; the buffer lies outside the designated segment.
0x00000107	ERR_GE_BUFFER_TOO_BIG	An invalid buffer address was specified: the buffer extends beyond the end of the segment.
0x00000108	ERR_GE_BAD_SELECTOR	Invalid buffer address; the selector is invalid.

## File and Pipe Access Errors

0x00000201	ERR_FS_INVALID_OFFSET	An invalid file offset was specified.
0x00000202	ERR_FS_NOT_FILE_OR_PIPE	The specified session number is not a file or a pipe.
0x00000203	ERR_FS_INVALID_OPTION	An invalid option was specified.
0x00000204	ERR_FS_NOT_KEYED_FILE	The specified session number is not for a keyed file.
0x00000205	ERR_FS_STRING_TOO_BIG	One or more strings is longer than 508 bytes.
0x00000207	ERR_FS_CONTROLLER_PIPE	An attempt to create or open a controller pipe occurred.
0x00000208	ERR_FS_NULL_STRING	One or more of the specified strings has a null value.

## Device Access Errors

0x00000301	ERR_PO_BAD_DEVICE_NAME	An invalid device name was specified.
0x00000302	ERR_PO_NO_IST	I/O Processor open error; no input sequence table was specified.
0x00000303	ERR_PO_BAD_PORT	Serial I/O open error: invalid port number specified. This value must be in the range of 1 to 4.
0x00000304	ERR_PO_BAD_PARITY	Serial I/O open error: invalid parity specified. Permitted values are E, O and N.
0x00000305	ERR_PO_BAD_SPEED	Serial I/O open error; invalid speed specified. Permitted values are 110, 300, 1200, 2400, 4800 , 9600 and 19200.
0x00000306	ERR_PO_BAD_STOP_BITS	Serial I/O open error; an invalid number of stop bits was specified. Permitted values are 1 and 2.
0x00000307	ERR_PO_BAD_DATA_BITS	Serial I/O open error; an invalid number of data bits was specified. Permitted values are 5 to 8.
0x00000308	ERR_PO_BAD_CLOSE_FNUM	A close was requested for an inactive session number.
0x00000309	ERR_PO_BAD_ROW_COL	Invalid row and column specified.
0x0000030A	ERR_PO_NOT_IOP_OR_MSR	This function is valid only for MSR or IOPROC.
0x0000030B	ERR_PO_NOT_TOTRET	This function is valid only for Totals Retention.

0x0000030C	ERR_PO_BAD_LENGTH	Invalid length specified for this device.
0x0000030D	ERR_PO_READ_NOT_VALID	This device cannot be read.
0x0000030E	ERR_PO_WRITE_NOT_VALID	Writing to this device is not supported.
0x0000030F	ERR_PO_BAD_HANDLER_ADDR	Defining an asynchronous error handler; invalid code pointer.
0x00000310	ERR_PO_NOT_DI_OR_CR	This function is valid only for the DI: and CR: printer stations.
0x00000311	ERR_PO_NOT_PRINTER	This function is valid only for the printer.
0x00000312	ERR_PO_BAD_SERIAL_BUFFLEN	Serial I/O: invalid buffer length specified.
0x00000313	ERR_PO_INVALID_OPTION	Reset parm on adx_clears is not 0 or 1.
0x00000314	ERR_PO_BAD_OFFSET	Hard Totals: invalid offset specified.

## Pipe Routing Service Errors

0x00000401	ERR_R_BAD_PIPE_ID	An invalid pipe ID was specified.
0x00000402	ERR_PR_NOT_PRS_PIPE	The specified session number is not a Pipe Routing Service pipe.
0x00000403	ERR_PR_BAD_DESTINATION	An invalid destination was specified.
0x00000404	ERR_PR_BAD_LENGTH	Invalid data length. This value must be in the range 1->120, if reading, or 1 -> 512, if writing, than 120 bytes.
0x00000405	ERR_PR_WRITE_ERROR	A write error occurred.
0x00000406	ERR_PR_DEST_NOT_FOUND	The specified destination was not found.
0x00000407	ERR_PR_NOT_PRS_DRIVER	The specified session number is not for the Pipe Routing Services driver.

## Application Service Errors

0x00000501	ERR_AS_BAD_FUNC_NUM	The function number specified is not supported in the terminal.
0x00000502	ERR_AS_BAD_BUFF_LEN	Invalid buffer length specified.
0x00000503	ERR_AS_BAD_DISK_ID	Cannot open input file.
0x00000504	ERR_AS_DISK_NOT_CONFIG	RAM or hard disk is not configured.
0x00000505	ERR_AS_DRIVER_NOT_CONFIG	The Application Services driver is not configured.
0x00000506	ERR_AS_BAD_TIME_PERIOD	An invalid time period was specified for adx_twait.
0x00000507	ERR_AS_BAD_EVENT_COUNT	An invalid event count was specified for adx_twait.
0x00000508	ERR_AS_BAD_EVENT_TYPE	An invalid event type was specified for adx_twait. Waits can be performed only on pipes or devices.
0x00000509	ERR_AS_BAD_INP_FNAME	An invalid input file name was specified.
0x0000050A	ERR_AS_INP_FILE_IN_USE	Input file is in use.
0x0000050B	ERR_AS_INP_FILE_ABSENT	Cannot find the input file.

0x0000050C	ERR_AS_INP_OWNER_DIFF	An ownership conflict occurred while attempting to access the input file.
0x0000050D	ERR_AS_INV_DEV_FOR_INP_FILE	An invalid device was specified for the input file.
0x0000050E	ERR_AS_OUT_FILE_IN_USE	The output file is in use.
0x0000050F	ERR_AS_INV_DEV_FOR_OUT_FILE	An invalid device was specified for the output file.
0x00000510	ERR_AS_OUT_FILE_EXISTS	The output file already exists and the replace option was not specified.
0x00000511	ERR_AS_OUT_FILE_NO_SPACE	There is insufficient disk space for the output file.
0x00000512	ERR_AS_OUT_FILE_WRITE	An error occurred while writing to the output file.
0x00000513	ERR_AS_INP_FILE_READ	An error occurred while reading the input file.
0x00000524	ERR_AS_INP_FILE_CLOSE	An error occurred closing the input file.
0x00000525	ERR_AS_OUT_FILE_CLOSE	An error occurred closing the temporary output file.
0x00000526	ERR_AS_OUT_FILE_DELETE	An error occurred deleting the temporary output file.
0x00000527	ERR_AS_OUT_FILE_RENAME	An error occurred while renaming the temporary file.
0x00000528	ERR_AS_BAD_OUT_FNAME	An invalid output file name was specified.
0x00000529	ERR_AS_BAD_OPTION	Invalid values specified for option parameters.
0x0000052A	ERR_AS_NOT_ENOUGH_MEM	There is insufficient memory available to complete the operation.
0x0000052B	ERR_AS_CLOSE	Error closing RAM or hard disk driver.
0x0000052C	ERR_AS_INVALID_OPTION	An invalid option was specified.
0x0000052D	ERR_AS_PARTIAL_WRITE	Only a partial write of data successful on call to <code>adx_terror</code> .
0x0000052E	ERR_AS_SESSION_RANGE	The specified session number is out of range.
0x0000052F	ERR_AS_BAD_INP_STRING	A zero length input string was passed to <code>adx_tcrypt</code> .

## Miscellaneous Functions

0x00000701	ERR_MI_BAD_PROGRAM_NAME	<code>adx_tchain</code> : an invalid program name was specified.
0x00000702	ERR_MI_NOT_ENOUGH_MEM	<code>adx_tmalloc</code> : there is insufficient memory available to satisfy the request.
0x00000703	ERR_MI_BAD_ADDRESS	<code>adx_tfree</code> : an invalid address was specified.
0x00000704	ERR_MI_PIPE_CLOSE	<code>adx_tchain</code> : error closing parameter passing pipe.
0x00000705	ERR_MI_PIPE_READ	<code>adx_tuse</code> : error reading arguments from parameter passing pipe.

0x00000706	ERR_MI_PIPE_READ	adx_tchain: error writing arguments to parameter passing pipe.
0x00000707	ERR_MI_PIPE_DELETE	adx_tuse: error deleting parameter passing pipe.
0x00000708	ERR_MI_PROGRAM_LOAD	adx_tchain: error loading program.
0x00000709	ERR_MI_CORRUPT_LENGTH	The address provided to adx_tfree is invalid (the leading length indicator is bad).
0x0000070A	ERR_MI_NOT_IN_USE	The address passed to adx_tfree has not been previously allocated to the application.
0x0000070B	ERR_MI_BAD_ARGLIST	Adx_tchain: Argument list is NULL, but a non-zero number of arguments has been specified.
0x0000070C	ERR_MI_ARGS_TOO_LARGE	adx_tchain: the total size of all the arguments exceeds 64K.
0x0000070D	ERR_MI_UNPACK_TOO_BIG	Adx_tunpack: Size of unpacked string too large to fit into a segment.
0x0000070E	ERR_MI_ZERO_SIZE_REQUEST	Adx_tmalloc: Zero heap size requested.
0x0000070F	ERR_MI_DSEG_TOO_SMALL	Data segment is too small to initialize the heap. Be sure DATA value used in linking the application is large enough (...DATA[max[fff]]...)

---

## Chapter 20. Designing applications with Java

This chapter discusses OS4690 Java support and describes Java classes in these packages:

Package	Description
---------	-------------

<b>com.ibm.OS4690</b>	
-----------------------	--

	Package for classes that provide access to various operating system features on store controllers and terminals. These features include POS files, keyed files, pipes, system status, and system controls.
--	--

<b>com.ibm.OS4690.jiop.util</b>	
---------------------------------	--

	Java I/O processor package for utility classes, such as System Monitor and Debug.
--	---

<b>com.ibm.OS4690.jiop</b>	
----------------------------	--

	Package for classes that provide Java I/O processor function.
--	---

For the operating system, the Java environment variable, CLASSPATH, is set through System Configuration. In Java 2, all system classes that are needed by the JVM are located automatically and read in by the JVM. Any entries in the "boot" classpath are searched prior to any user classes that are specified on the command line or through the CLASSPATH environment variable. The Java 2 classpath should contain only user classes. Any existing references to the Java 1.1.8 JVM system classes, such as CLASSES.ZIP, SWINGALL.JAR, and S4690.JAR should be removed for Java 2. See the <http://docs.oracle.com/javase/1.4.2/docs/tooldocs/findingclasses.html> Web site for more information on how the Java 2 JVM locates classes.

**Note:** Java 1 is no longer supported with this version of the OS. Existing Java 1.1.8 or Java 2 applications that have been configured or put into batch files may need changes to run on Java 1.6. To run these Java 1.1.8 or Java 2 applications under Java 1.6, you must explicitly change their configuration or batch files, as well as the applications themselves where File() class methods are used.

---

### Setting the classpath in Java 2

There are several methods to set the classpath environment variable for the Java 2 JVM. Because Java 2 JVM is compiled using the VisualAge C/C++ compiler, its environment variables can be set like any other VisualAge program. These are the methods to set the classpath for the Java 2 JVM:

- Set the classpath related logical names through the 4690 system configuration. See the *4690 OS: Planning, Installation, and Configuration Guide* for more information about setting the classpath using this method. This method sets several system logical names JAVA2CP1 through JAVA2CP8, which are concatenated together to form the classpath, with the unneeded ones set to empty values. If the process logical name JAVA2CP1 is set then that set of logical names will be used instead.
- Set the logical name CLASSPATH to the desired value. This method will override the classpath set via configuration (JAVA2CP#). Setting this logical name will affect Java 6 as well.
- Set the CLASSPATH environment variable in one of the ADX\_SDT1:ADXENV.DAT files as described in Chapter 22, "Creating 32-Bit Programs Using VisualAge C/C++," on page 665. Doing this overrides any value that has been set in the classpath related logical names.
- Use the `-classpath` or the `-cp` command line parameters. This method overrides all other methods of setting the classpath.

This is the default controller classpath:

```
javali:os4690.zip;
```

This is the default terminal classpath:

```
r::java\lib\os4690.zip;r::c:\java\jpos4690.zip  
r::c:\java\jpos14.jar;r::c:\java\jpos4690.zip;  
r::c:\java\ibmjpos.jar;
```

**Note:** Do not copy any of the Java 2 programs, DLLs, or other files from the locations where they are currently installed unless you fully understand the results. The Java 2 JVM uses the location of its executable programs to locate additional DLLs, class files such as RT.JAR, fonts, language extensions, and other configuration files. Java 2 JVM programs might not work correctly if files are renamed or moved from their current location.

---

## Setting the classpath in Java 6

There are several methods to set the classpath environment variable for the Java 6 JVM.

**Note:** The methods are identical to the ones used when setting the classpath environment in Java 1.4.2 JVM, except the method for using VisualAge environment variables. Because Java 2 JVM is compiled using the VisualAge C/C++ compiler, its environment variables can be set like any other VisualAge program.

These are the methods to set the classpath for the Java 6 JVM:

- Set the classpath related logical names through the 4690 system configuration. See the 4690 OS: Planning, Installation, and Configuration Guide for more information about setting the classpath using this method. This method sets several system logical names JAVAECPP1 through JAVAECPP8, which are concatenated together to form the classpath, with the unneeded ones set to empty values. If the process logical name JAVAECPP1 is set then that set of logical names will be used instead.
- Set the logical name CLASSPATH to the desired value. This method will override the classpath set via configuration (JAVAECPP#). Setting this logical name will affect Java 2 as well.
- Use the -classpath or the -cp command line parameters. This method overrides all other methods of setting the classpath

There is no default controller classpath.

There is no default terminal classpath.

---

## Using response files to start Java programs on a terminal

In many cases, the length of the parameter list (including properties, JVM flags, and class parameters) needed to start the Java applications can become quite long. Lengthy parameter lists can occur when using Java Terminal Offline Function (JavaTOF) to support offline operations on the terminal. A “response file” must be used when the length of the parameter list exceeds the 64-character limit imposed by the Java terminal application configuration. A response file allows the list of parameters to be specified in a file located on the terminal RAM disk.

When a Java program is started on the terminal (either as a primary Java application or from the Start Java Application screen in OCF), the first line of the response file is read in and passed to Java as its parameter string. The response file can contain both JVM options flags (such as -Xxxxx -Dxxxx, -g, and so on) as well as the user class name and parameters. Currently, the length of the response file is limited to 8K.

To use a response file, perform the following steps:

1. Create a file containing the flags and parameters. Only the first line of the file is processed and any additional lines are ignored.
2. Add the file to the terminal RAM disk preload list.
3. Modify the Java parameter list in the terminal load definition and specify @fn.rsp as the parameter list, where fn.rsp is the name of the response file. By default, the response file is looked for on the first

- terminal RAM disk (X:). If you want to place the response file on the second RAM disk, prefix the response file name with the drive letter, for example @y:/fn.rsp, where y represents the drive letter.
4. Apply terminal configuration and reload the terminal.

The response file must be placed on the terminal RAM disk, particularly when JavaTOF is used, to ensure the program load happens with as few controller file accesses as possible. Accessing a response file located on the controller can severely impact the load time performance of multiple terminals.

---

## Java application support

With appropriate application support, .ZIP files used by the terminal could also be located in a terminal RAM disk or on available terminal DASD. The classpath would then point to the appropriate files. See the *4690 OS: Planning, Installation, and Configuration Guide* for information on configuring classpaths.

**Note:** Due to the dynamic class loading, care must be taken when you encounter a situation at the terminal where an attempt is made to load a class that cannot be found due to a loss of communication with the controller.

The operating system allows you to configure a batch file as a primary or secondary application, which allows a Java application to be run from the 4690 menu. For example, the batch file, ADX\_UPGM:JAVAPRIM.BAT, could contain the line `java SalesApplication` and the primary application name would be configured as ADX\_UPGM:JAVAPRIM.BAT. Selecting the primary application from the menu would start Java with the `SalesApplication` class. Other supported batch file commands are also allowed. Secondary applications can use batch files in a similar manner. The command session ends when the configured batch file completes. If a nonzero value is returned from this command shell, 4690 treats this response as an error and displays "Application ended abnormally." If a nonzero return is normal for your application, add an additional command in your batch file so when it ends, zero is returned to 4690. For example, the batch file, JAVAPRIM.BAT, could contain:

```
java SalesApplication
REM my application returns non-zero,
REM issue ECHO OFF to clear errorlevel
ECHO OFF
```

If the program started is Java-based, the operating system allows Java to be configured as a background application. To configure Java in the background, enter `JAVAEBIN:JAVA.386` (or `JAVA2BIN:JAVA.386`) as the program name with the classname as a parameter. Unlike other background programs, `BACKGRND` is not passed to Java as the first parameter. Instead, the system property, `BACKGRND=TRUE`, is set so that the Java application can determine that it is running in the background. For example:

```
String bg = System.getProperty("BACKGRND");
```

assigns the string value `TRUE` to `bg`, if Java was started in the background.

---

## Using Java 6

Java 6 at the Java Runtime Environment (JRE) level 1.6 is supported by 4690 Version 6 Release 2 or higher when operating in Enhanced Mode. JRE level 1.6 is mostly compatible with Java programs written and compiled with earlier Java levels. Any incompatibilities between various Java levels are documented on Oracle's Java web site <http://docs.oracle.com/javase/6/docs/>. Java 6 support is automatically installed on Enhanced Mode controllers on 4690 OS Version 6 Release 2 and higher. For information on designing an application with Java, see the *4690 OS: Programming Guide*. To run a Java program with Java 6, use the program `javaebin:java.386`. The `javaebin:` logical name refers to the directory `/adxetc/java/bin` on the enhanced drive F:. The directory `F:/adxetc/java` contains other directories that are used with Java 6. These directories include:

- `bin` — This is where Java 6 related programs are located. Currently, only `java.386` is included.
- `core` — `javacore.txt` and `heapdump` files will be placed here.



- `home` — This is the default location of the user's home directory (the value of the `user.home` property is set to this location).
- `lib` — Any core JAR files are placed here. Currently this includes the files required to run RMA.
- `tmp` — This is the default location of the JVM's temporary directory (the value of the `java.io.tmpdir` property is set to this location).
- `lib/ext` — Java extension files are listed here. This directory is added to the `java.ext.dirs` property in the JVM.

Compared to the Java 2 support in 4690, some functionality and capabilities are not present in the Java 6 JVM:

- Multi-App is not supported with Java 6.
- No Software Development Kit (SDK) functions are included at the 1.6 level, the only SDK functions are available as part of the 1.4.2 SDK. This includes such tools as `jar`, the applet viewer and the `java` compiler.
- A "debug" version of the JVM (`java_g`) is not included.
- The only JRE program that is directly accessible is `java.386`.
- User written Java Native Interface (JNI) libraries are not supported.

## Starting Java 6

Running the program `javaebn:java.386` without parameters will provide some information about the supported JVM parameters. The Oracle website referenced above provides more detailed information, although some of the flags are Oracle-specific. When browsing 's documentation, follow the links for the **Linux** version of the tools, which more accurately describes the capabilities of the JVM. For more precise documentation on this JVM, you may download the documentation at the at Developer Works web site. Select the JVM titled "J2SE Version 6.0 (32-bit xSeries (Intel compatible))" and download the file in TGZ format (gzipped tar file). The Java SDK download contains a directory called `ibm-java2-i386-60/docs` which contains information specific to this JVM; start by viewing the file `startHere.htm` in a web browser.

When any program is started in 4690, it uses the concept of a "current directory" to determine the location of files without absolute paths. This value is set in the logical name default:. Typically the logical name is set to a drive letter (which is another logical name) that is used to determine the directory on that drive. The Java 6 JVM uses this same concept except that the current directory of the JVM is always somewhere on 4690 F: drive. If the user's current directory is located on the F: drive when the JVM is started, then its current directory will be set to that location. Otherwise the JVM's current directory will be set to the root of the F: drive.

Note that the F: drive supports long file names but is different from other 4690 drives such as C: or M: in that the filesystem is case sensitive. For example, a file named `FILE.DAT` can coexist on the F: drive with a file named `file.dat`. Thus, attempting to run Java 6 with the program name `JAVA.386` will not work. Use care in both naming and referencing files placed on the F: drive. The F: drive is available both for FTP and NFS access, both of which allow for case-sensitive filenames.

## Properties

To facilitate writing portable programs that will work either on Java 2 or Java 6, and to allow Java 6 programs to detect that they are running on OS4690, several properties are always set. These properties are set in Java 2 JVM as well.

- `com.ibm.OS4690.os.version` — Set to the version and release of OS4690 (i.e. "6.2")
- `com.ibm.OS4690.os.mode` — This is set to a number indicating whether 4690 is running in classic (0) or enhanced (1) mode.
- `com.ibm.OS4690.os.level` — Set to the base and CD levels of OS4690. Also includes a descriptive string indicating the OS mode. i.e. "Base(09A0) CD(09B0) mode(0:Classic)"



- `com.ibm.OS4690.adxeloop` — The loopback IP address (127.0.0.1) and the machine's external IP address do not work properly when they are used to communicate between native 4690 programs and Java 6 (the traffic is not passed between them). This property is set to an IP address that must be used when writing a Java 6 program that needs to communicate to a native 4690 program on the same machine. A logical name `ADXELOOP` is set to the same value for use by non-Java programs.

## Writing Java 6 Programs

Although Java is supposed to be "write once, run anywhere," there are differences between Java 2 and Java 6 that can cause problems with applications. Some of the differences are mentioned in this and other sections and are specific to how each JVM is implemented for OS4690. Please reference the following Oracle websites for more details on general differences between the JVM levels:

<http://www.oracle.com/technetwork/java/javase/index-jsp-138567.html>

<http://www.oracle.com/technetwork/java/javase/overview/index-jsp-136246.html>

One important thing to note is that the Java 6 JVM is Linux based. It runs in a Linux service layer that allows it to share information with other programs running on OS4690, however it does not have access to the all OS4690 resources. As discussed above, the F: drive is one of the resources that is shared with OS4690. From the point of view of most OS4690 programs, there is a case sensitive disk drive called F: that can be used to store files. From the point of view of Java 6, the F: drive is a directory in a Linux-based file system with the path `/cdrive/f_drive`. To prevent users from having to hard-code this value, the property `com.ibm.OS4690.path.f_drive` is set with the location of the F: drive for a Java 6 program. Other properties starting with "`com.ibm.OS4690.path.`" are defined when these paths are available, however the preferred method of determining the linux path name of a 4690 file is to use the method `com.ibm.OS4690.File4690.getJvmPath()`. This method resolves any 4690 logical names in a path (e.g. "`ADX_SPGM:`") that doing simple string manipulation will not.

Other important rules and conventions of the Linux operating system (and Linux-based JVMs) are as follows:

- The `os.name` and `os.version` properties will report the name and version of the underlying Linux service layer. See below for information on how to determine the version of OS4690.
- The default user name is "`vxuser`".
- The Java 6 JVM does not use the normal 4690 C runtime, thus any environment variables normally used to configure or control the Java 2 JVM do not work.
- The default locale information and timezone of the JVM are set based on the `LC_ALL` and `TZ` logical names configured and setup during OS4690 boot. Setting these logical names at runtime (i.e as process logical names) will not affect the JVM. The java system properties set by these logical names may be overridden from the Java command line if needed.
- All files must be accessed with paths relative to the current directory or absolute paths. For example, the file `F:/adxetc/java/lib/os4690.zip` contains the `IBMDefault` java class. To run `IBMDefault` from a command prompt, the commands will work equally well. They assume the user's current directory in 4690 is "F:".
 

```
javaabin:java -cp adxetc/java/lib/os4690.zip IBMDefault
javaabin:java -cp /cdrive/f_drive/adxetc/java/lib/os4690.zip IBMDefault
```
- A security manager is installed by default on Java 6. The `java.security` property may not be overridden.
- The security manager prevents access to other portions of the Linux file system. A user program may only access files in `/cdrive/f_drive`. See below for information on how to access 4690 files.
- The Java 6 JVM cannot run 4960 programs directly. The security manager prevents the user from running Linux programs (via `Runtime.exec()`). See below for information on how to run 4690 programs.
- The file separator for Java 6 is the forward slash (`/`). Using a backslash (`\` or ASCII character 92) will not work. In Java 2 using either one was valid.
- The path separator for the classpath is a colon (`:`) not a semicolon (`;`).

There are some other environmental differences between Java 2 and Java 6 of which you may need to be aware of:

- There is no default classpath set for the 1.6 JRE for either Enhanced controllers or Enhanced terminals, but classpath entries may be set in System Configuration.
- The JIT (just in time compiler) is enabled by default for Java 6. Setting the java property `java.compiler` to `NONE` will disable it.
- If a Java 6 program enters graphical mode (creates a window), the console screen will be replaced with a graphical desktop (as with Java 2) known as the **enhanced mode graphical extensions screen**. All windows created by that JVM or any other Java 6 JVMs will appear on that screen. In addition, windows created by other Linux programs, like MBrowser, will appear on this screen as well. A navigation bar is provided at the bottom of the screen to allow navigation between windows. When **Alt-SysReq** is used to navigate among windows, the **enhanced mode graphical extensions screen** will be displayed whenever the JVM window is displayed. However, the JVM window may not be shown *on top* of the other windows.
- When using `TerminalApplicationServices` class to select the mode used for ANPOS keyboards, a Java6 application must ensure that the keyboard is set to POS mode only when it has focus. If the keyboard remains in POS mode when another application (such as MBrowser) has focus, the keyboard will not provide input to the application. See the section titled "Switching the terminal Java keystroke destination for shared ANPOS keyboards" on page 529 for more information.

The `stdin/stdout/stderr` streams for Java 6 behave similarly to how they do in Java 2. By default output appears on the console screen or can be sent to a file with command shell redirection or other methods as with Java 2. Ctrl-C works as it does on 4690 (ending the JVM).

## Access to OS4690 resources and programs

The Java 6 JVM can directly access only files on the 4690 F: drive (i.e. files within `/cdrive/f_drive`). This includes the file classes such as `FileInputStream`, `FileOutputStream`, `RandomAccessFile`, and `File`. Programmatic file access to files on other 4690 drives, such as C: or M:, and file paths using logical names, are provided using new file classes that are the same, but with "4690" at the end. These classes include `FileInputStream4690`, `FileOutputStream4690`, `RandomAccessFile4690`, and `File4690`.

As discussed above, the `Runtime.exec()` function is restricted and cannot run 4690 programs. The class `Runtime4690` can be used to run 4690 programs in a manner similar to the Java `Runtime` class.

The classes required for 4690 resource access are packaged in `f:/adxetc/java/lib/os4690.zip` as well as in `javaliib:os4690.zip`. For portability the classes may be used either in Java 2 or Java 6. Documentation for the new classes added in OS4690 V6 is located in `4690opt/adxejava.jar` on the installation CD.

---

## Internationalization and DBCS enablement in Java

It is relatively easy to design Java programs such that they can be used in a variety of languages and locales without requiring code changes. In a typical Java program, this design is done for all languages that need to be supported, not just DBCS languages. However, supporting DBCS languages can be more difficult than supporting Latin-based languages, particularly for programmers that are not familiar with DBCS languages. The reason for this difficulty is because with DBCS languages, you cannot depend on Latin-based concepts, such as using a space to delimit words in a sentence or sorting strings based on their Unicode character values.

For more information on internationalization, see the following sources:

- The "Internationalization" section of the Java 2 SDK Standard Edition Documentation at <http://docs.oracle.com/javase/1.4.2/docs/guide/intl/index.html>
- The "Internationalization Trail" in the Java Tutorial at <http://docs.oracle.com/javase/tutorial/i18n/index.html>

- The "Internationalization" section of the JDK 6 Documentation at <http://docs.oracle.com/javase/6/docs/technotes/guides/intl/index.html>

---

## Locales and encodings in Java

Locales are used in Java to help abstract country and region information from a program's implementation. The initial default locale in Java is determined by the values of the `user.language` and `user.region` system properties. The default value of these properties is determined by the setting of the `LC_ALL` environment variable, which itself is set based on how 4690 OS was configured during installation. For more information on the locales supported by Java, see the <http://docs.oracle.com/javase/1.4.2/docs/guide/intl/locale.doc.html> Web site.

Many of the Java classes support the conversion of bytes to characters and back again. These conversions are done by "pluggable" character converters. The `file.encoding` system property is used to determine the default converter used by the Java I/O and other classes. The default value of the `file.encoding` property is determined by the current locale. If a different encoding is needed, it can be overridden through a command line switch. In addition, many Java methods that deal with character conversion have versions that allow you to specify the encoding to use. An example of this is the `getBytes` method of `java.lang.String`. For more information on the encodings supported by Java, see the <http://docs.oracle.com/javase/1.4.2/docs/guide/intl/encoding.doc.html> Web site.

---

## Fonts in Java 2

The 4690 OS Java 2 implementation uses font rendering technology based in part on the work of the FreeType team. For more information on FreeType technology, see the <http://www.freetype.org> Web site.

## Installing additional fonts

**Note:** The user cannot install additional Fonts for Java 1.6.

On most platforms, new font files can be installed in either the operating system's standard location for fonts or within the directory structure of the Java JRE or SDK. The 4690 OS platform does not have a "standard location" for fonts; therefore, by default, fonts are looked for only in the `m:/java2/jre/lib/fonts` directory. If the system property `java.awt.fonts` is defined, the Java VM also looks for fonts in the path names that are given in the property's value. The default value of this property is retrieved from the environment variable, `JAVA_FONTS`.

Several fonts are automatically installed in the Java 2 SDK's `jre/lib/font` directory. Most of these fonts are default fonts provided by Oracle. However, there are several fonts that are unique to 4690. The following fonts are installed for all language builds and are contained in the glyphs in the WGL4 font encoding:

**MonotypeSansDuospaceWT.ttf**

(Monospaced sans serif)

**MonotypeSansWT.ttf**

(Proportional sans serif)

**ThorndaleDuospaceWT.ttf**

(Monospaced serif)

**TimesNewRomanWT.ttf**

(Proportional serif)

In addition, DBCS TrueType fonts are shipped for each appropriate DBCS language build. Four additional font files are shipped for each DBCS language. The font encodings are "East Asia" encodings, which means that Arabic, Hebrew, and Hindi glyphs were removed to improve character spacing. The names of the font files are identical to the names of the WGL4 fonts listed above, except that the name of the font file has a language-specific suffix. These are the suffixes which are used for the DBCS languages that are supported by 4690 OS:

- JEA — Japanese (East Asia encoding)
- KEA — Korean (East Asia encoding)
- SCEA — Simplified Chinese (East Asia encoding)
- TCEA — Traditional Chinese (East Asia encoding)

For example, this is the list of files that is shipped with the Japanese language build of the 4690 OS:

- TimesNewRomanWTJEA.ttf
- ThorndaleDuospaceWTJEA.ttf
- MonotypeSansDuospaceWTJEA.ttf
- MonotypeSansWTJEA.ttf

## Loading DBCS fonts for use by terminal applications

Due to their size, the DBCS versions of these fonts are not transferred when the Java 2 JVM is loaded to the terminal M or Q drive. If you require DBCS fonts on a terminal, create a preload bundle containing the following file specifications:

- m:/java2/jre/lib/fonts/TimesNewRomanWT\*EA.ttf
- m:/java2/jre/lib/fonts/MonotypeSansDuospaceWT\*EA.ttf
- m:/java2/jre/lib/fonts/MonotypeSansWT\*EA.ttf

You do not need to select any of the checkboxes when adding these paths to the bundle. The paths will match the DBCS fonts for the DBCS language you have installed (if any). If you do not require all of these fonts, then you can include only the specific fonts needed. For example if all DBCS text is displayed in the "dialog" font (which is a proportional sans serif font) then you only need to include MonotypeSansWT\*EA.ttf.

After creating the bundle, add it to the load definition for your java application. It should be extracted to the same drive to which the JVM is being preloaded. For more information see the *4690 OS User's Guide* chapter titled "Enhanced terminal preload and Java Configuration".

## The font.properties files

It is not necessary to edit the font.properties files in order to use new fonts. A specific font can be specified by directly using its embedded TrueType font name, such as "Monotype Sans WT". You need to edit the font.properties files only if you want to map one of the logical font types, such as Serif, Sansserif, or Monospaced, to a new font or if locally-specific mapping is needed.

Several font.properties files are included with the Java 2 SDFK and are located in the directory m:/java2/jre/lib. The font.properties files contain information used by Java to render fonts for display. The default font.properties file has no suffix on its file name. Like other properties files in Java, locale- and regions-specific properties can be given by appending the locale name to the property file name.

In the Japanese (ja), Korean (ko), Chinese (zh), or Traditional Chinese (zh\_TW) locales, the font.properties file that corresponds to that particular locale is used instead of the default font.properties file. The font.properties files are identified by the country or local suffix that is appended to the file name.

font.properties.<locale> where <locale> is one of the following options:

```
ja
ko
zh
zh_TW
```

Editing font.properties files is not supported. However, there might be circumstances where it is necessary to edit font.properties files to achieve a particular result. In those circumstances, edit the font.properties files and select the applicable font to the locale in which you are running. For example, for Traditional Chinese, select the font.properties.zh\_TW font file. Be aware that this file is part of the JRE and affects all

Java 2 applications on both the controller and on all terminals that are loaded from the controller. For more information on editing font properties files, see the <http://docs.oracle.com/javase/1.4.2/docs/guide/intl/fontprop.html> website,

The font.properties files are part of the JVM and new copies are shipped along with any JVM maintenance. Therefore, any user modifications to these files are lost whenever OS maintenance is applied, including JVM changes. Keep backup copies of all user modifications to ensure that these modifications are not lost or replaced when OS maintenance is applied.

## DBCS and Latin character alignment

The Java font, "monospaced", is typically used to display fixed-width characters. When a mixture of Latin and DBCS characters are displayed, the DBCS characters are typically twice the width of the Latin characters. This additional width allows multiple lines to be displayed vertically and still line up correctly. This alignment can be done in order to emulate a printer receipt or 2x20 display. By default, a Java application using the "monospaced" font to do this type of multiline display does not look correct because the characters are misaligned. The reason for this misalignment is because Java 2 treats "monospaced" as a composite font. A composite font means that Java 2 retrieves font glyph outlines from several different font files and treats them as one font. The reason for doing this is that Java 2 allows smaller fonts that align better to be used and yet still be able to display all of the needed characters. As an example, see the following section from the file, font.properties.ja:

```
monospaced.0=Lucida Sans Typewriter Regular
monospaced.1=Monotype Sans Duospace WT
monospaced.2=Monotype Sans Duospace WT J EA
```

```
monospaced.bold.0=Lucida Sans Typewriter Bold
monospaced.bold.1=Monotype Sans Duospace WT
monospaced.bold.2=Monotype Sans Duospace WT J EA
```

When attempting to display a plain, monospaced character, the JVM first looks for that character in the font "Lucida Sans Typewriter Regular." If the character is not there, then JVM looks in the two Monotype fonts (in order). The bold, monospaced font displays using this same logic except that the first font is a Lucida font that is specifically designed as a bold font. Even though characters from the Monotype font can be made to look bold (algorithmically), the Lucida fonts typically look better (especially for small typefaces).

The composite font behavior causes misaligned text because any Latin characters in a string come from the Lucida font and any DBCS characters come from the Monotype font. Even though the characters might be of the same height, the font metrics of these two fonts are different. Therefore, the Latin glyphs are not exactly half of the width of the DBCS characters (as expected).

If you are able to change the fonts used by the program, there are two ways of getting the characters to correctly align:

- Specify the exact name of the font that you want to use. The exact font name is the TrueType name as embedded in the font file and listed in font.properties. To use a sans-serif, monospaced font with a Japanese language build, you could use "Monotype Sans Duospace WT J EA" as the name. The Java method call `GraphicsEnvironment.getLocalGraphicsEnvironment().getAllFonts()`; can be used to return a list of all installed fonts in Java 2.
- A better method for specifying one of the built-in fonts is to use one of the built-in pseudo names. The font.properties files, shipped with the 4690 OS, contain four unique font names that map to each of the four WorldType fonts installed with the JVM. The default font properties file maps the fonts as follows:

```
wtserif.0=Times New Roman WT
wtsans.0=Monotype Sans WT
wtmonoserif.0=Thorndale Duospace WT
wtmonosans.0=Monotype Sans Duospace WT
```



In the DBCS font properties files, these names are mapped to the corresponding DBCS fonts installed in that OS language build. For example, in the Japanese language build, `wt sans` refers to "Monotype Sans WT J EA." Therefore, if you are creating an application to be used under several different language builds, you can use one of the pseudo font names in your program and have the correct DBCS font selected at runtime.

If you cannot change (or it would be difficult to change) the application, the `font.properties` file can be edited so that `monospaced` is an alias to the specific font you want to use. An alias is specified by adding a property of the form `"alias.fromName=toName"`, where `"fromName"` is the font your program uses and `"toName"` is the actual font name or a reference to another font in the `font.properties` file. For example, to prevent Lucida from being used for the monospaced font in a Japanese language build, add the following line to `font.properties.ja`:

```
alias.monospaced=Monotype Sans Duospace WT J EA
```

A better option could be to use the line:

```
alias.monospaced=wtmonosans
```

This option causes the `monospaced` alias to refer to one of the pseudo fonts in the file, as explained above. Because this option is non-locale specific, the same line can be added to each of the font properties files and still refer to the correct WorldType font for each of the different DBCS language builds.

## Fonts and JavaTOF

**Note:** JavaTOF is not supported with Java 1.6.

When using the JavaTOF function, all needed class files as well as resources and fonts are packaged into the terminal loadshrink file to be placed on the terminal RAM disk. See Chapter 14, "Using the Java Terminal Offline Function," on page 249 for additional JavaTOF information. The `font.properties` file, appropriate to the current locale, is used to determine which fonts to include in the loadshrink file. When running on DBCS language builds, the `font.properties` file, by default, lists all four DBCS fonts. The DBCS fonts are very large due to the number of glyphs that are contained in them. The large size of the DBCS fonts can cause the creation of the loadshrink file to fail or can cause the RAM disk image to be too large. To work around this problem, edit the appropriate `font.properties` file (for example, `font.properties.ja` for Japanese) to remove the fonts that are not needed.

Although four complete DBCS fonts files are shipped, there are only two distinct font faces in these files: Serif (Mincho) and SansSerif (Gothic). Because the DBCS glyphs are all the same width, there is no "proportional" font face. For example, the `ThorndaleDuospaceWTJEA.ttf` and `TimesNewRomanWTJEA.ttf` font files both contain the Serif (Mincho) form of the DBCS glyphs. The main difference between these two files is the style of the Latin glyphs (proportional or monospaced).

In order to reduce the size of the loadshrink file, determine which of the four DBCS fonts you do not need. Typically, an application might use only one of the font faces, Mincho or Gothic, for displaying data. In this case, choose the font according to the style of any Latin characters you are also displaying. For example, if you want to emulate a 2x20 customer display, you would probably want to use a monospaced Gothic font because the SansSerif fonts are easier to read at a distance and you would want the Latin and DBCS characters to align correctly on both lines of the display. In this case, choose the font, `MonoTypeSansDuospaceWTXXX.ttf`, where `XXX` is the appropriate language suffix.

In some cases, both Mincho and Gothic typefaces might need to be used in your terminal application. In those cases, choose the two best fonts (one Mincho and one Gothic) and include only those two fonts in the loadshrink file. If you are displaying any monospaced Latin characters in your application, you would probably want to use the two font files that have monospaced Latin characters: `ThorndaleDuospaceWTXXX.ttf` and `MonoTypeSansDuospaceWTXXX.ttf`, where `XXX` is the appropriate language suffix.

After you have determined the fonts that are not needed, then you must edit the appropriate, locale-specific font.properties file and remove the references to those font filenames. In addition, you need to adjust references to those fonts so that Java is able to use an alternate font to locate DBCS characters. The easiest way to accomplish this is to perform the following steps:

1. Remove any file name references to the fonts that you want to remove. JavaTOF determines what gets placed into the loadshrink by looking at the properties beginning with "filename." In the original properties files, these file name references are at the end of the file. Comment out or delete the lines referring to the fonts that you want to remove.
2. Replace any references to the fonts you just deleted with references to other similar fonts. For example, if you removed the "proportional" DBCS fonts, "Times New Roman WT J EA" and "Monotype Sans WT J EA", then you need to locate and change any references to those fonts to the other similar ones. In this case, you would change the name "Times New Roman WT J EA" to "Thorndale Mono Duospace J EA" and you would change the name "Monotype Sans WT J EA" to "Monotype Sans Duospace WT J EA." If you are removing all but one of the DBCS fonts, change any references to the three fonts you removed to the one font that remains. Note that the **case** and **spacing** of the font names are important; therefore, be careful when changing the names.
3. Optionally, if the application is having alignment problems when displaying monospaced text that is a mixture of Latin and DBCS characters, see the above section. One or more of the following lines might need to be added to the font properties file:

```
alias.monospaced=wtmonosans
alias.courier=wtmonosans
alias.dialoginput=wtmonosans
```

---

## Java limitations on 4690 OS

**Note:** JavaTOF is not supported with Java 1.6.

### File name limitations

There are several file name restrictions that should be noted when porting Java applications from other platforms:

- In 4690 OS Version 6 or later, the F: drive is available in enhanced mode. The drive supports long file names but is different from other 4690 drives such as C: or M: in that the F: drive is case-sensitive. F: drive filenames may be 8.3 or 'long file names'. The drive letter is only available from native OS4690 programs. The Java 6 JVM accesses the contents of the F drive as part of a Linux filesystem. This is discussed in more detail in the "Using Java 6" section below.
- Programmatic file access supported by the Java 6 for files on other 4690 drives, such as C: or M:, and so on, is provided using new file classes that are the same, but with "4690" at the end, such as `FileInputStream4690`, `FileOutputStream4690`, `RandomAccessFile4690`, `File4690`, and so on. All the classes required for file access are packaged in `f:/adxetc/java/lib/os4690.zip`.
- 4690 OS file name support on the non-VFS and non-NFS drives is restricted to the PC-DOS style 8.3 format. The JVM silently truncates longer filenames internally to this format when both creating and opening files. Therefore, care should be taken when porting applications from other platforms to ensure that external file names remain unique after this truncation occurs.
- The VFS file system supports filenames up to 256 characters in length.
- The file name support on drives mounted through the NFS file system is dependent on the capabilities of the NFS server and of the remote drive.
- Files accessed using RIOAM (R::) are restricted to the 8.3 format. In addition, the total path length of files, including the directory names, accessed through RIOAM cannot exceed 26 characters.
- Due to a restriction in the RIOAM interface, directories cannot be created remotely on the controller using RIOAM (R::). If the creation of a valid directory is attempted, it appears to succeed, but instead a 0 length file of the same name is created.

## Setting a default directory for Java applications on terminals

Generic Java applications can be written to read files from the current, or the default, directory. Because a terminal typically does not have a local file system, there is no concept of a current directory when running on the terminal. To allow such applications to run correctly, the current directory for terminals is set by a logical name. This logical name is TJAVA2DEF for Java 2 and TJAVADEF for Java 6. The terminal application loader reads this logical name during initialization and sets the default directory to the value that it contains.

The setting of the default directory can be used in several different ways in a Java program. For example, it is used when user code attempts to create or open a disk file using a relative (nonabsolute) file name. In addition, the value of the Java system property, `user.dir`, contains the name of the user's current directory at the time that the JVM was started (although, this value can be overridden through the command line or programmatically).

The default value of the TJAVA2DEF logical name is `java2bin:.../.../`, which is resolved to the root directory of the M: drive on the controller where Java 2 was loaded from. Normally, this is located on the master controller. However, if the ADXJ2TPTH logical name was set (allowing Java 2 to be loaded from alternate NFS drives), the current directory for Java applications could be located on other controllers.

The default value of the TJAVADEF logical name is `F:/`, which resolves to the linux path `/cdrive/f_drive`. The logical name must be set to a valid 4690 path name, for example, `f:/adxetc`. If the logical name does not resolve to a location on the F: drive, the current directory of the given drive will be changed. In this case, it will affect the operation of the File4690 APIs but the JVM itself will be `/cdrive/f_drive`.

In order to change the default current directory, change the TJAVA2DEF or TJAVADEF logical name as appropriate. To change the default current directory, create the file, `ADX_IDT1:ADXTRMUF.DAT`, if it does not exist, and add a line to it containing the new setting, for example:

```
TJAVA2DEF=L:/Home/  
TJAVADEF=f:/adxetc/java/home/
```

Either one or both names can be changed and the names can be the same or different. The new name must end in either a directory separator character (`/`) in either JVM or a colon (`:`) for Java 2 to ensure that it is used correctly. If TJAVA2DEF is changed to make use of RIOAM (R:), be aware of the restrictions with this approach as listed above. After making these changes, the new terminal configuration must be applied and the terminals reloaded for the change to take effect.

If your application requires the current directory to always be available, it should be set to a location such as the terminal RAM disk, in case communication with the controller is lost. If you want to refer to the controller in the path setting, then use `R::` (which remains valid even if controller backup is done) or use a path relative to the `java2bin:` logical name (as the default does), in case ADXJ2TPTH is being used.

## Thread-dispatching differences

The 4690 OS does not prevent "starvation" during scheduling of threads. A Java thread with a higher priority that runs continuously prevents any other Java threads with a lower priority from running. This fact should be taken into consideration when writing an application that sets thread priorities. This affects Java 2 threads only.

## Setting the user's home directory on the terminal

In Java, the value of the `user.home` property is defined as "the user's home directory." Because the 4690 OS does not have a concept like this, the `user.home` property for Java 2 is set to the "root" of the Java installation tree. The "root" of the Java installation tree in Java 2 is `m:/java`. When running in a terminal, this value is set to `javabin:.../...`, respectively. In Java 6 the home directory is set to `f:/adxetc/java/home`.



**Note:** In a 4690 OS V6 Enhanced Mode terminal, the "root" of the Java installation tree is Java 6 /cdrive/f\_drive.

The following methods can be used to override these values in order to point to a different home directory:

- Use the command line switch, -D, to change the value of the system property, for example:

```
-Duser.home=my_directory
```

This method is the preferred method and works with both JVMs.

- For Java 2, set the C run-time environment variable, JAVAHOME, to the value you want to use. This value is identical for both controllers and terminals. To set the C run-time environment variable, create the file, ADX\_SDT1:ADXENV.DAT, if it does not exist, and add a line to it containing the new setting, for example:

```
JAVAHOME=your_value
```

Chapter 22, "Creating 32-Bit Programs Using VisualAge C/C++," on page 665 contains more information on how to set C run-time environment variables. How to create different settings for controllers than for terminals is also explained in this chapter.

A similar Java property, java.home, is defined to be the location of the Java installation directory. Currently, the default value of this property is set automatically based on where the JVM was loaded from and it should not need to be modified.

## Java problem resolution

Table 52. Java problem resolution

Symptom	Solution
After installing or applying maintenance, the system seems hung at W555:J92.	This message is displayed while files are being extracted to the M: and F: drives. This can take a fairly long time if there is a lot of data to extract (particularly if large MBCS fonts are being installed).
When I run a Java 6 program from the command line, it still seems to run in Java 2 mode.	Make sure that you prefix the Java command with <b>JAVAEBIN:</b> .
Java programs that use graphics, such as create AWT or swing components, do not work. Text mode Java applications such as IBMDefault work fine.	For the controller or terminal that you are running the application on, make sure that Java graphics are enabled.
It looks like Java 6 works, but my programs do not run. (For terminal programs, the system event log does not seem to indicate any problems other than the fact that JAVA.386 ends.)	<p>First, look at the Java console screen for that terminal and check to see if any error messages are displayed. You may need to redirect stderr to see these messages. If error messages are displayed, they might help in determining what is wrong.</p> <p>Second, if this action does not fix the problem, check the following items:</p> <ul style="list-style-type: none"><li>• The command line parameters for Java 6 are different than for Java 2. Make sure the parameters you used are correct for Java 6. Running Java with no arguments displays a list of valid parameters.</li><li>• Make sure that the JAR file or directory containing your classes is listed in the classpath and that the correct path for Java 6 is used, including the proper case.</li></ul>
My Java programs run fine on pure terminals but my classes are not found when run on a controller terminal.	Remember that the <b>controller's</b> classpath and drive letters are used by Java programs running on the terminal side of a controller terminal. In this case, the JAR file or directory containing your classes must be listed in the controller's classpath.

Table 52. Java problem resolution (continued)

Symptom	Solution
My Java programs run fine on pure terminals but the graphics screen resolution and/or colors are incorrect when run on a controller terminal.	<p>For Java programs running on the terminal side of Classic controller terminal, the Primary and Secondary video graphics screen resolution and color format are configured in <b>Controller Configuration &gt; Video Attributes</b>.</p> <p>For Java programs running on the terminal side of an Enhanced controller terminal, the Primary video graphics screen resolution and color format are configured in <b>Controller Configuration &gt; Video Attributes</b>. The Secondary video graphics screen resolution and color format are configured in <b>Terminal Configuration &gt; Device Characteristics &gt; Video Displays &gt; Graphics</b>.</p>
When running Java Applets with Java 6, I get a security exception when the applet code tries to do certain tasks, such as read files. The applet works fine in Java 2.	The applet security model changed between Java 2 and Java 6.
The fonts in my graphical application (AWT/Swing) are a lot smaller in Java 2/Java 6 than in Java 1.	<p>The initial Java 1 code shipped with 4690 did not correctly handle font sizes. In this older code, screen font sizes (pixel heights) were chosen assuming 96 dpi (dots per inch) screen resolution. Java, however, assumes a 72 dpi screen resolution. With this older code, a 12-point font on 4690 appears much larger on screen compared to other platforms. Typically, application programmers coded around this problem by choosing smaller fonts.</p> <p>This problem was fixed in later versions of Java 1 for 4690. However, to allow current Java 1 programs to run unmodified, a method was provided to switch back into "old fonts" mode (running with the -Dof flag). The -Dof flag is not supported in new JVMs because the font sizes are already correct. In order for your application to run correctly in V6, the font sizes that the application uses need to be increased.</p>
I am trying to run the AppletViewer (sun.applet.Appletviewer) and it is saying that r: is an unknown protocol.	Newer JVMs appear to be stricter about making sure that the URL passed to the AppletViewer is valid. If the file name passed to the program contains a colon, the value before the colon is assumed to be the transfer protocol. Running the AppletViewer on Windows NT with a name such as <i>c:\test.htm</i> gives similar results. To avoid this problem, prefix the file name with <i>file</i> to form a proper URL.

Table 52. Java problem resolution (continued)

Symptom	Solution
A Java program ends suddenly with a message similar to "***Out of memory, exiting***". You have coded an exception handler for the OutOfMemoryError in your Java code and the exception never seems to be generated or caught.	<p>This message occurs when the JVM attempts to allocate memory for "native" run-time objects (those data not stored in the Java heap). Typically, these native objects are allocated in response to allocations of Java objects. In low memory situations, the type of failure seen depends on where the error is detected. If the out of memory condition is detected when the Java heap is being expanded or if the heap is expanded to its maximum limit, as defined by the <code>-mx</code> parameter, then an OutOfMemoryError is thrown by the JVM. If the out of memory condition is detected in other areas of the JVM, it might not be able to continue running and exits with an error message.</p> <p>The solution to this problem is to add memory to the system or to reduce memory usage of the Java program. If you are using the <code>-mx</code> or <code>-Xmx</code> parameters to set the maximum size of the Java heap to a large value, try using a smaller value.</p>
When using JavaTOF on a DBCS system to allow a terminal application to be used in offline mode, the resulting loadshrink file is too large.	There is a limit to the size of the loadshrink file. Normally, this limit is large enough to handle almost any reasonable application. However, when running Java in DBCS mode, the very large DBCS font files are also included in the loadshrink file. This problem can be resolved typically by editing the font.properties file and removing the fonts that you do not need. See "Fonts in Java 2" on page 461 for additional information on fonts and how to prevent JavaTOF from including unnecessary files.
I need to start a Java application on a terminal, but need to specify more than 64 characters for the JVM flags, class, and parameters.	See "Using response files to start Java programs on a terminal" on page 456 for detailed instructions.
An application that is being shown on the secondary display of a Dual Display system is requesting input (such as through a modal dialog) and will not continue until input is provided. There is no way to provide this input because there is no available keyboard, mouse, or touch device.	<p>You would normally be careful to select and/or program the applications that are placed on the secondary display to ensure that the application does not require input from the user. However, in some cases, an application that normally would not require input, for various reasons, does require input. Typically, the method for resolving this problem is specific to each application.</p> <p>An example of this problem is with the AppletViewer. The first time that the (Java 1) AppletViewer is run on a system, the AppletViewer requires the user to accept a license agreement by clicking on a button. The AppletViewer no longer requires this action after the first time the license agreement is accepted on a system. If the first time the AppletViewer is run on a system is on a secondary display with no input devices, it is not possible to click on a button. In this particular example, the solution is to run the AppletViewer the first time on a primary terminal display or on the controller and accept the license agreement.</p>

Table 52. Java problem resolution (continued)

Symptom	Solution
I want my Java 2 application to appear on the secondary display.	Unless explicitly configured otherwise, the default display for a Java 2 application is always the Primary video display. The multi-app function is required to configure an application's default display to be the secondary video display. For information on how to configure the default display using multi-app, refer to the "Startup Property File" section in the <i>4690 OS: Programming Guide</i> .
I want my Java 6 application to appear on the secondary display.	<p>For Java 6, you must use a Java property to configure the default display for an application. The property is <code>com.ibm.OS4690.edisplay</code>. The valid values for this property are:</p> <ol style="list-style-type: none"> <li>1 Use the Primary display as the default display for an application.</li> <li>2 Use the Secondary display as the default display for an application.</li> </ol> <p>By default, when the Java console or application window is displayed, the contents of both displays is switched to the Java console simultaneously. If the display number above is followed by the lowercase letter <code>o</code>, then only the default display is swapped. For example, to select the secondary display as an application's default display add the flag <code>-Dcom.ibm.OS4690.edisplay=2</code> to the JVM options of the application command line. Note that the Secondary video display does not support all of the functions that are available on the Primary video display. Touch input is supported, but input from a mouse or keyboard is not supported on the Secondary video display. Any pointing device that is attached to the terminal will affect the cursor on the primary display, including a mouse attached to any USB ports on the secondary display.</p>
I want my Java application to use both displays.	<p>Access to multiple displays is only supported for terminal-based Java 2 applications. For Java 2, this functionality is possible using the methods in the <code>java.awt</code> package to query the local graphics environment and get a list display devices. The default configuration may be queried from each device and used to create a window or dialog that is shown on the given display. Refer to the method</p> <pre>java.awt.GraphicsEnvironment.getScreenDevices()</pre> <p>as a starting point.</p> <p>Access to multiple displays is not supported in Java 6. The default display for a Java 6 application may be chosen as indicated in the previous Solution.</p>

Table 52. Java problem resolution (continued)

Symptom	Solution
In a graphical Java program, there are <i>boxes</i> displayed where there should normally be blanks or other characters displayed.	Java displays these <i>boxes</i> whenever the user program attempts to display invalid characters, such as those characters for which the selected font does not define a glyph. The value and shape of the undisplayable character is determined by each specific font file, but typically, it is displayed as a rectangular <i>box</i> . In many cases, a program appeared to work in Java 1.3.0 but does not work in later Java versions. This difference is caused by a change in the way that some characters are handled (for example, the null character, "\u0000"). In earlier versions of Java (such as 1.3.0), the null character was not displayed or was displayed as a blank. But in Java 1.4.1 and later, the null character is treated as an invalid character and is displayed as a rectangular <i>box</i> . This behavior is consistent with JVMs on other platforms (such as Microsoft Windows).
I am using the logical name to allow for Java 2 NFS backup (i.e., ADXJ2TPTH). However, when the primary controller is offline the Java application loads very slowly or appears to hang.	Make sure that none of the paths or filenames used by the Java program reference the NFS drive letter for the primary controller (L:). This includes components of the classpath, the boot classpath, and locations of property files. It also includes values for these components that are contained within the JavaTOF or MultiApp properties files.
I am using JavaTOF in an attempt to "preload" everything to the terminal for rapid access. However, I am still seeing many file accesses to the controller after the terminal is loaded. Also, if the controller goes down, my application behaves abnormally.	Ensure that all references to files made by your application are to files on the terminal RAM disk. You might also want to check the Java text console, which is displayed when the application ends, for any JavaTOF warning messages. See Chapter 14, "Using the Java Terminal Offline Function," on page 249 for additional information about these messages. Also, ensure that any drive letter references in the command line, in the JavaTOF/MultiApp/application property files, and in the terminal response file are to files on the terminal. If your application refers to drives or files located in the default, user home, or Java home directories, you might need to move the files to the RAM disk and let the application refer to them there (by changing the appropriate property or the application). If your application must access files on the controller, the application must be able to handle retries and reopening of the file if the controller goes down. Using files on RIOAM (R::) allows this to happen in a more automated fashion (such as the file is reopened if it is already open, but the application still needs to retry the open, in case an offline error is returned). In any case, an application should be careful not to do file I/O on the AWT or Swing event threads, otherwise, the UI hangs until the request times out.

Table 52. Java problem resolution (continued)

Symptom	Solution
I am attempting to resolve a Java application problem. I tried running the application with the -verbose JVM flag. In addition to all the class loaded messages, I see a lot of exceptions being generated. Is this the cause of the problem?	<p>These exceptions might or might not be related to the problem. Exceptions are displayed in the verbose output whenever native (non-Java) code signals the exceptions back to the JVM. Exceptions created and thrown within Java code are not displayed.</p> <p>The exceptions that are displayed could help identify the problem, but not all exceptions necessarily indicate a problem. When the JVM loads native code DLLs, it uses the search path given by the java.library.path property (which is by default the 4690 DLL search path) and calls a method to load the DLL from each location. The JVM catches the internal exceptions that get thrown from this call as it searches each of the paths. If it is still not successful, it generates and throws a new exception. A typical user DLL loaded by this method will typically cause several exceptions to be displayed in the verbose output before the DLL load succeeds.</p>

## Frequently asked Java questions

- **Where can I find information about Java 6?** The best place is to start with Oracle's <http://www.oracle.com/technetwork/java/javase/overview/index-jsp-136246.html> website.
- **Where can I find information about Java 2?** The best place is to start with Oracle's <http://www.oracle.com/technetwork/java/javase/overview/index.html> website.
- **How do I develop JNI (native) code for Java 2 on 4690?** Oracle's documentation provides the general details for how to create and build JNI code for Java 2. In order to compile and link JNI code for a specific platform, you need the header (.h) files and link libraries for that platform. For 4690, the header files and link libraries are located on the 4690 controller in the directories m:\java2\include and m:\java2\lib, respectively.

## Additional hints and tips

- The following classes are for use with Java 1 only and were not shipped with OS4690 V6, **do not** add them to your classpath: classes.zip, swingall.jar, s4690.jar.

**Note:** os4690.zip can be used with both Java 2 and Java 6. However, it **can only** be used on 4690. It contains various classes and methods that are used to access 4690-specific functionality.

- In order to reduce the amount of memory required by the JVM, the number of ZIP and JAR archives on the classpath should be minimized. This statement is applicable for both Java 2 and Java 6. Upon initialization, the JVM constructs lookup tables for each archive so that the JVM can look for class definitions in those archives without opening and searching the archives for every search. The amount of memory required for each archive depends on how many entries that archive contains. An archive with several thousand entries requires several hundred KB of memory to contain the lookup table. For example, RT.JAR contains over 5000 files and requires approximately 200 KB of memory for the lookup table.
- If you frequently run Java programs manually, note that you can use `-cp` as a shorthand version of the `-classpath` flag with the JAVA.386 program. This shorthand version works for Java 2 and Java 6.
- As their default, terminals run the IBMDefault class as their default Java application. This class displays information about the OS and JVM environment making it easier to determine what version of Java a terminal is using. By passing the parameter "info" to the program, additional information is displayed. By passing the parameter "gui" to the program, a graphical window is displayed containing the additional information.

- By default, in Java 2 the Java Just In Time Compiler (JIT Compiler) is disabled in order to reduce memory usage. To enable the JIT Compiler, you can:
  - Set the `java.compiler` property to `jtc` (**-Djava.compiler=jtc** on the command line).
  - Set the environment variable, `JAVA_COMPILER`, to `jtc`. See Chapter 22, “Creating 32-Bit Programs Using VisualAge C/C++,” on page 665 for information on setting C run-time environment variables.
- By default, in Java 6 the Java Just In Time Compiler (JIT Compiler) is enabled. To disable the JIT Compiler, you can:
  - Set the `java.compiler` property to `NONE` (**-Djava.compiler=NONE** on the command line).
  - Set the environment variable, `JAVA_COMPILER`, to `NONE`.
- How to tell the difference between Java 2 and Java 6:
  - Programmatically: Examine the System property `java.version`. It is set to 1.4.2 for Java 2, and to 1.6.0 for Java 6.
  - Visually: Unless overridden, the background color of the title bar of Frames and JFrames is blue in Java 2 and blue in Java 6, but with a bottom frame bar that is also blue.
  - Visually: The system menu for Java 2 windows contains an additional entry used to select the input methods. In English locales, the additional entry is "Select Input Method."
- Java 2 supports the TrueType fonts. Additional user fonts can be installed in the `m:\java2\jre\lib` directory. If you install fonts from a third party, you must ensure that you follow any licensing requirements required for those fonts. Additional fonts may not be installed for Java 6.
- In order for Java 2 to work on terminals, the following items must be configured:
  - VFS must be configured on the controller.
  - Both the controller and the terminal must be configured to use TCP/IP.
  - NFS must be configured and running on the controller (NFS Server, Portmapper, and PCNFSD, if required).
  - The `M:\` drive on the controller must be exported by the controller (listed in `ADX_SDT1:ADXHSIXF.DAT`).
  - The NFS configuration for the terminal must mount the `M:\` drive of the controller as the `L:\` drive of the terminal.
- In order to use Java 2 on the controller, you must have VFS configured for the `C:` drive.

**Note:** For Enhanced Mode controller VFS is enabled for `C:` by default.

- For V6 Enhanced Mode controllers, you do not need to prefix the Java command with `JAVA2BIN`, but this prefix is still recognized. For Java 1.6, be sure to prefix the Java command with `JAVAEBIN`: when running a Java 6 program from the command line.
- Make sure that the JAR file or directory containing your classes is listed in the Java 2 or Java 6 classpath, depending upon the Java version you intend to run.
- Because the command line parameters for Java 2 are different from Java 6, ensure the parameters you used are correct for the version of Java that you are using to launch your application.
- When compiling your Java applications, ensure that you are using a Java compiler that does not add unexpected language extensions to the class file, for example, `J++`.
- If you are developing applications that will run on either Java 2 or Java 6, it is suggested that you use Java 2 to compile them. This ensures that you will only use APIs and language features that are available in both. Alternatively, you may use the `-target` and `-bootclasspath` compiler flags (or a development environment that does the same thing) to target a specific JRE level.
- To use Java code within the 4690 OS environment:
  1. Create ZIP or JAR files of your Java code.
  2. Copy the files to the 4690 controller (remembering to do a `chkdsk -F`, if necessary).
  3. Put the names of the ZIP or JAR files in the classpath from the SYSTEM CONFIGURATION panel.
- Ensure that the full path name is a maximum of 25 characters for each entry in the terminal classpath, if you are using `R::` to access the files. The 25-character limit includes the `R::` characters.



- On a terminal with a Java application loaded, pressing **Alt-SysReq j** or **Alt-SysReq t** before the terminal is fully loaded results in an inability to access the Java console. Java runs, but you are not able to access it.
- On a controller, you cannot toggle between a Java window and the command line that started the Java application. To return to that specific command prompt, the Java program must be exited.
- All printing from Java must be done using a PostScript printer.
- Creating deeply nested subdirectories can cause the LAN Disk Rebuild utility, ADXNSL0L, to abend. For versions of the 4690 operating system prior to Version 5, the LAN Disk Rebuild Utility could not handle complete file names that were longer than 47 characters. For example, ADXLXTCN::H0://ADX\_UDT1/LEVEL1/LEVEL2/FINLEY11.TXT exceeds the limit beginning with TXT. Version 5 adds support for complete file names that are a maximum of 128 characters, which is the limit set by the OS shell.
- If a terminal Java application indicates that a file is not found on the controller but the file does exist, make sure that the terminal application is requesting the file in an R::XXXXXXXX:XXXXXXXX.XXX format. The file name should be a maximum of 25 characters including the R:: characters. If the file name is too large, the controller indicates that the file cannot be found.
- When running TCP/IP applications in the controller, it is recommended that you define the logical name, HOSTNAME, to the value of the TCP/IP host name of the controller. Java provides a method called `getLocalHost()`, which returns the TCP/IP host name of the controller on which the Java application is running. This method returns results that are not valid if the HOSTNAME logical name is not defined.

---

## Direct serial and parallel port communication

The operating system provides a COMM4690.JAR package, which is its implementation of JAVAX.COMM. This package is contained in C:\JAVA\COMM4690.JAR. This file must be included in the terminal classpath if any application uses classes in the JAVAX.COMM package. The file must be transferred to the terminal or referred to via R:: (for Java 2) so the terminal can access it. The COMM4690.JAR package is valid only on the terminal.

The COMM4690.JAR package supports the use of bi-directional communication through serial ports and unidirectional communication through parallel ports. The OS4690 parallel port driver supports the writing and outputting of data. The unidirectional Standard Parallel Port (SPP) mode is supported. The EPP, ECP, Nibble, and PS/2 modes are not supported.

The API for both the serial port and the parallel port is documented on Oracle's [http://docs.oracle.com/cd/E17802\\_01/products/products/javacomm/reference/api/index.html](http://docs.oracle.com/cd/E17802_01/products/products/javacomm/reference/api/index.html) website.

**Note:** On a controller/terminal, the JAVAX.COMM parallel port can be used from the terminal side if the controller side is not using the parallel port.

These are the classes and interfaces involving the parallel port:

- `javax.comm.CommPort`

These are the methods implemented and supported from this class for the parallel port:

- `close`
- `toString`
- `getName`
- `getOutputBufferSize`
- `getOutputStream`
- `setOutputBufferSize`

- `javax.comm.ParallelPort`

All of the methods for this class are implemented and supported. The `setMode` accepts only `LPT_MODE_SPP` or `LPT_MODE_ANY` as valid input. The port operates only in SPP mode.

- `javax.comm.ParallelPortEvent`

All of the methods for this class are implemented and supported.



- `javax.comm.ParallelPortEventListener`

The `parallelPortEvent` method for this class is implemented and supported.

The `COMM4690.JAR` package, when communicating through serial ports, differs from the `JAVAX.COMM` package in several ways:

- The Carrier Detect methods are not supported.
- The Ring Indicator methods are not supported.
- The output buffer is limited to 247 bytes. Beginning with Version 6 Release 3, the output buffer limit is 1024 bytes. However, not all serial devices can support larger buffers. The user is responsible for assuring that the buffer size is appropriate for the attached serial device.
- Receive framing is not supported.
- The following baud rates are supported:
  - 110
  - 300
  - 1200
  - 2400
  - 4800
  - 9600
  - 19200

Beginning with Version 6 Release 3, the following additional baud rates are also supported:

- 14400
- 28800
- 38400
- 57600
- 115200

**Note:** To set the baud rate to 115200, specify the value as 1152.

- `STOPBITS_1_5` are unsupported stop bits.

**Note:** To configure for 1.5 stop bits, use 5 data bits and 2 stop bits.

- `PARITY_SPACE` and `PARITY_MARK` are unsupported parities.
- These are unsupported flow controls:
  - `FLOWCONTROL_XONXOFF_IN`
  - `FLOWCONTROL_XONXOFF_OUT`
- `FLOWCONTROL_RTSCCTS_OUT` is always set. `FLOWCONTROL_RTSCCTS_IN` can be set by the application.

#### Notes:

1. Both RS-232 ports and device channel ports emulating RS-232 ports are supported.
2. Logical names for the ports use the `COMn` naming convention, where *n* is the logical number (1–4) to which a port has been configured.

The operating system provides these classes:

- `FlexosException`
- `InvalidParameterException`
- `POSFile`
- `KeyedFile`
- `POSPipeInputStream`

- POSPipeOutputStream
- ControllerApplicationServices
- ControllerStatusData
- multiapp.ApplicationManager
- TerminalApplicationServices
- TerminalStatusData
- TerminalKeyboardLights

I/O redirection to Java classes includes:

- DeviceManager
- ANDisplayHandler
- IOHandler
- CashReceiptMonitor
- ANDisplayHandlerException
- IOHandlerException

Java I/O processor classes in package com.ibm.OS4690.jiop include:

- JIOProcessor
- IOPReadyListener
- IOPReadyEvent
- IOPInitStatus
- PromptChangeListener
- PromptChangeEvent
- IOPShutdownListener
- SystemBusyListener
- SystemBusyEvent
- IOPInputQueue
- QueueStatusChangeListener
- QueueStatusChangeEvent
- POSKeyListener (deprecated)
- POSKeyListenerEx
- LabelInputListener
- LabelInputEvent
- QueueLockedException
- StateForbidsInputException
- StateTableProcessor
- StateChangeListener
- StateChangeEvent
- State
- FunctionCode
- Globals
- InputSequenceFormatter
- InputSequenceClearedListener
- InputSequenceClearedEvent
- FieldActivatedEvent
- FieldDataUpdatedEvent

- FieldDeactivatedEvent
- FullScreenAttributes  
VideoParms
- FullScreenFieldListener
- FunctionCodeList
- InputFieldListener

Java I/O processor classes in package `com.ibm.OS4690.jiop.util` include:

- SystemMonitor
- ExceptionListener
- ExceptionEvent

---

## Java Application Programming Interfaces

### Class FlexosException

This section explains the constructors and methods for the FlexosException class. Many methods within other classes throw FlexosException exceptions if there is an error.

```
public class com.ibm.OS4690.FlexosException
    extends IOException
{
    //Constructors
    public FlexosException(int)
    public FlexosException(int, String)

    //Methods
    public int getReturnCode()
}
```

### FlexosException Constructors

This section contains the constructors for the FlexosException class.

#### ***FlexosException(int):***

```
public FlexosException(int retcode)
```

#### **Parameters:**

##### **retcode**

The FlexosException return code

#### ***FlexosException(int, String):***

```
public FlexosException(int retcode,
    String DetailString)
```

#### **Parameters:**

##### **retcode**

The FlexosException return code

##### **DetailString**

The detailed message returned

### FlexosException Methods

This section contains the method for the FlexosException class.

#### ***getReturnCode():***

```
public int getReturnCode()
```

Specific return codes are described in the methods that expect to throw FlexosException exceptions.

### Class InvalidParameterException

The operating system contains a Java class to return a detailed description if an incorrect parameter is entered as part of a method.

```
public class com.ibm.OS4690.InvalidParameterException
    extends IOException
{
    //Constructor
    public InvalidParameterException(String)
}
```

## InvalidParameterException Constructor

This section contains the constructor for the `InvalidParameterException` class. This constructs an `InvalidParameterException` exception with the specified detail message.

```
public InvalidParameterException(String DetailString)
```

### Parameters:

#### DetailString

The detailed message returned

## Class POSFile

The operating system supports POS files, which are random-access files that can be distributed to other controllers as mirrored or compound files. The operating system provides a `POSFile` class that you can use to create and access files that have distribution attributes. Once you create a distributed file, you can use other methods in this class to access the file. You can also use the `java.io` core classes to open and access the file if file record sharing and locking capabilities are not required. The constructors that create a POS file cannot be used in a terminal.

The methods in this class can be used to provide reading and writing restrictions for other instances of this class that represent the same file.

```
public class com.ibm.OS4690.POSFile
    extends Object
    implements Def4690
{
    //Fields
    public static final short COMPOUND_FILE
    public static final byte DISTRIBUTE_ON_CLOSE
    public static final byte DISTRIBUTE_ON_UPDATE
    public static final int EXCLUSIVE_ACCESS
    public static final short FLUSH
    public static final short FROM_CURRENT_FILE_POINTER
    public static final short FROM_END_OF_FILE
    public static final short FROM_START_OF_FILE
    public static final short LOCK_EXCLUSIVE
    public static final short LOCK_SHARED
    public static final short LOCK_WRITE
    public static final short MIRRORED_FILE
    public static final short NO_FLUSH
    public static final short NO_WAIT_ON_LOCK
    public static final short READ_FROM_CACHE
    public static final short READ_FROM_DISK
    public static final int SHARED_READ_ACCESS
    public static final int SHARED_READ_WRITE_ACCESS
    public static final short WAIT_ON_LOCK

    //Constructors
    public POSFile(File, String, int)
    public POSFile(File, String, int, int)
    public POSFile(File, String, int, int, short, byte)
    public POSFile(String, String, int)
    public POSFile(String, String, int, int)
    public POSFile(String, String, int, int, short, byte)

    //Methods
    public void closeFull()
    public void closePartial()
    protected void finalize()
    public int getFilePointer()
    public int length()
    public void lock(int, int, int, int, int)
    public int read(byte[], int, int, int, int)
    public void seek(int)
```

```

    public int skipBytes(int)
    public void unlock(int, int, int)
    public void write(byte[], int, int, short, int)
    public void writeAppend(byte[], int)
}

```

## POSFile Fields

This section explains the fields for the POSFile class.

### ***COMPOUND\_FILE:***

```
public static final short COMPOUND_FILE
```

This field indicates a file type. A read-only image of the file on the file server resides on all other controllers.

### ***DISTRIBUTE\_ON\_CLOSE:***

```
public static final byte DISTRIBUTE_ON_CLOSE
```

This field is a distribution indication; distribute the file when the file is closed.

### ***DISTRIBUTE\_ON\_UPDATE:***

```
public static final byte DISTRIBUTE_ON_UPDATE
```

This field is a distribution indication; distribute the file when the file is updated.

### ***EXCLUSIVE\_ACCESS:***

```
public static final int EXCLUSIVE_ACCESS
```

This field indicates an access restriction. Only this object is allowed read and write access.

### ***FLUSH:***

```
public static final short FLUSH
```

This field forces the data to the disk.

### ***FROM\_CURRENT\_FILE\_POINTER:***

```
public static final short FROM_CURRENT_FILE_POINTER
```

This field is a file pointer for offset reference.

### ***FROM\_END\_OF\_FILE:***

```
public static final short FROM_END_OF_FILE
```

This field is a file pointer for offset reference.

### ***FROM\_START\_OF\_FILE:***

```
public static final short FROM_START_OF_FILE
```

This field is a file pointer for offset reference.

### ***LOCK\_EXCLUSIVE:***

```
public static final short LOCK_EXCLUSIVE
```

This field indicates a lock restriction. Other POSFile objects cannot access the file record.

**LOCK\_SHARED:**

```
public static final short LOCK_SHARED
```

This field indicates a lock restriction. Other `POSFile` objects can read and lock the record.

**LOCK\_WRITE:**

```
public static final short LOCK_WRITE
```

This field indicates a lock restriction. Other `POSFile` objects have read-only access to the record.

**MIRRORED\_FILE:**

```
public static final short MIRRORED_FILE
```

This field indicates a file type. A read-only image of the file on the file server resides on the alternate file server.

**NO\_FLUSH:**

```
public static final short NO_FLUSH
```

This field does not force the data to the disk.

**NO\_WAIT\_ON\_LOCK:**

```
public static final short NO_WAIT_ON_LOCK
```

This field specifies lock conflict handling; do not wait on the lock to release.

**READ\_FROM\_CACHE:**

```
public static final short READ_FROM_CACHE
```

This field indicates to read the data from the disk cache.

**READ\_FROM\_DISK:**

```
public static final short READ_FROM_DISK
```

This field indicates to read the data from the disk.

**SHARED\_READ\_ACCESS:**

```
public static final int SHARED_READ_ACCESS
```

This field indicates an access restriction. Only this object is allowed write access to the file.

**SHARED\_READ\_WRITE\_ACCESS:**

```
public static final int SHARED_READ_WRITE_ACCESS
```

This field indicates an access restriction. Any object can access the file.

**WAIT\_ON\_LOCK:**

```
public static final short WAIT_ON_LOCK
```

This field specifies lock conflict handling; wait on the lock to release.

**POSFile Constructors**

This section describes the constructors for the class `POSFile`.

***POSFile(File, String, int):*** This constructor opens an existing POS file to read from and write to the file specified by the *file* parameter. The file pointer is initialized to the beginning of the file when the file is opened.

```
public POSFile(File file,
               String mode,
               int access) throws FlexosException, InvalidParameterException
```

**Parameters:**

**file**     The file object.

**mode**    Indicates to open the file for both input and output. The parameter must be:

**rw**     Read/Write

**access**

Indicates the file access restrictions. The parameter must be one of the following:

- EXCLUSIVE\_ACCESS
- SHARED\_READ\_ACCESS
- SHARED\_READ\_WRITE\_ACCESS

This constructor throws a *FlexosException* exception if the file cannot be opened. It throws an *InvalidParameterException* exception if an incorrect parameter is passed.

***POSFile(File, String, int, int):*** This constructor creates and opens a local POS file to read from and write to the file specified by the *file* parameter. This file is not distributed to the other controllers on the network. The file pointer is initialized to the beginning of the file when the file is opened. POS files cannot be created by a class running in a terminal.

```
public POSFile(File file,
               String mode,
               int access,
               int fileSize,) throws FlexosException, InvalidParameterException
```

**Parameters:**

**file**     The file object.

**mode**    Indicates to open the file for both input and output. The parameter must be:

**rw**     Read/Write

**access**

Indicates the file access restrictions. The parameter must be one of the following:

- EXCLUSIVE\_ACCESS
- SHARED\_READ\_ACCESS
- SHARED\_READ\_WRITE\_ACCESS

**fileSize**

Specifies the number of bytes to reserve for the file.

This constructor throws a *FlexosException* exception if the file already exists or cannot be created. It throws an *FlexosException* exception if an incorrect parameter is passed.

***POSFile(File, String, int, int, short, byte):*** This constructor creates and opens a distributed POS file to read from and write to the file specified by the *file* parameter. In this constructor, the file pointer is initialized to the beginning of the file when the file is opened. POS files cannot be created by a class running in a terminal.

```
public POSFile(File file,
               String mode,
               int access,
```



```

        int fileSize,
        short fileType,
        byte distributionMethod) throws FlexosException,
        InvalidParameterException

```

#### Parameters:

**file** The file object.

**mode** Indicates to open the file for both input and output. The parameter must be:

**rw** Read/Write

#### access

Indicates the file access restrictions. The parameter must be one of the following:

- EXCLUSIVE\_ACCESS
- SHARED\_READ\_ACCESS
- SHARED\_READ\_WRITE\_ACCESS

#### fileSize

Specifies the number of bytes to reserve for the file.

#### fileType

Indicates the file distribution type. The parameter must be one of the following:

- MIRRORED\_FILE
- COMPOUND\_FILE

#### distributionMethod

Indicates when the file is to be distributed. The parameter must be one of the following:

- DISTRIBUTE\_ON\_UPDATE
- DISTRIBUTE\_ON\_CLOSE

This constructor throws a `FlexosException` exception if the file already exists or cannot be created. It throws an `InvalidParameterException` exception if an incorrect parameter is passed.

***POSFile(String, String, int):*** This constructor opens an existing POS file to read from and write to the file with the specified name. The file pointer is initialized to the beginning of the file when the file is opened.

```

public POSFile(String name,
               String mode,
               int access) throws FlexosException, InvalidParameterException

```

#### Parameters:

**name** File name.

**mode** Indicates to open the file for both input and output. The parameter must be:

**rw** Read/Write

#### access

Indicates the file access restrictions. The parameter must be one of the following:

- EXCLUSIVE\_ACCESS
- SHARED\_READ\_ACCESS
- SHARED\_READ\_WRITE\_ACCESS

This constructor throws a `FlexosException` if the file cannot be opened. It throws an `InvalidParameterException` exception if an incorrect parameter is passed.

**POSFile(String, String, int, int):** This constructor creates and opens a local POS file to read from and write to the file with a specified name. This file is not distributed to the other controllers on the network. The file pointer is initialized to the beginning of the file when the file is opened. POS files cannot be created by a class running in a terminal.

```
public POSFile(String name,
               String mode,
               int access,
               int fileSize) throws FlexosException, InvalidParameterException
```

#### Parameters:

**name** File name. If the file is not to be created in the current directory, *name* must include the path specification. The maximum length is 128 characters.

**mode** Indicates to open the file for both input and output. The parameter must be:

**rw** Read/Write

#### access

Indicates the file access restrictions. The parameter must be one of the following:

- EXCLUSIVE\_ACCESS
- SHARED\_READ\_ACCESS
- SHARED\_READ\_WRITE\_ACCESS

#### fileSize

Specifies the number of bytes to reserve for the file.

This constructor throws a `FlexosException` exception if the file already exists or cannot be created. It throws an `InvalidParameterException` exception if an incorrect parameter is passed.

**POSFile(String, String, int, int, short, byte):** This constructor creates and opens a distributed POS file to read from and write to the file specified. The file pointer is initialized to the beginning of the file when the file is opened. POS files cannot be created by a class running in a terminal.

```
public POSFile(String name,
               String mode,
               int access,
               int fileSize,
               short fileType,
               byte distributionMethod) throws FlexosException,
               InvalidParameterException
```

#### Parameters:

**name** File name. If the file is not to be created in the current directory, *name* must include the path specification. The maximum length is 128 characters.

**mode** Indicates to open the file for both input and output. The parameter must be:

**rw** Read/Write

#### access

Indicates the file access restrictions. The parameter must be one of the following:

- EXCLUSIVE\_ACCESS
- SHARED\_READ\_ACCESS
- SHARED\_READ\_WRITE\_ACCESS

#### fileSize

Specifies the number of bytes to reserve for the file.

#### fileType

Indicates the file distribution type. The parameter must be one of the following:

- `MIRRORED_FILE`
- `COMPOUND_FILE`

#### **distributionMethod**

Indicates when the file is to be distributed. The parameter must be one of the following:

- `DISTRIBUTE_ON_UPDATE`
- `DISTRIBUTE_ON_CLOSE`

This constructor throws a `FlexosException` exception if the file already exists or cannot be created. It throws an `InvalidParameterException` exception if an incorrect parameter is passed.

## **POSFile Methods**

This section explains the methods for the class `POSFile`.

***closeFull():*** This method closes this file stream and releases any system resources associated with the stream.

```
public void closeFull() throws FlexosException
```

This method throws a `FlexosException` exception if a file error occurs.

***closePartial():*** This method is a partial close that flushes the buffers, but leaves the file open.

```
public void closePartial() throws FlexosException
```

This method throws a `FlexosException` if a file error occurs.

***finalize():*** This method ensures that a full close is done for this file stream when it is no longer referenced.

```
protected void finalize()
```

The method overrides *finalize* in class `Object`.

***getFilePointer():*** This method returns the current pointer offset into this file from the beginning of the file.

```
public int getFilePointer() throws FlexosException
```

This method throws a `FlexosException` exception if a file error occurs.

***length():*** This method returns the length of the file.

```
public int length() throws FlexosException
```

The method throws a `FlexosException` exception if a file error occurs.

***lock(int, int, int, int, int):*** Access to file records by other `POSFile` objects is restricted by locking and unlocking file records. Lock a record to restrict it; unlock the record to allow access. Typically, a record is locked before it is read and then unlocked after it is written back to the file. This method locks a record. The location of the record to be locked is determined by the file pointer. The file pointer is positioned to the offset value referenced from the `offsetReference` reference.

```
public void lock(int lockType,
                int lockConflict,
                int offset,
                int offsetReference,
                int length) throws FlexosException
```

#### **Parameters:**

##### **lockType**

Indicates the type of lock requested. The types are:

- LOCK\_EXCLUSIVE
- LOCK\_WRITE
- LOCK\_SHARED

#### **lockConflict**

Indicates whether to wait on a lock conflict. In a controller, if a record is locked by another object, you can either block waiting on that lock to be released or have a `FlexosException` exception thrown. The parameter can be:

- WAIT\_ON\_LOCK
- NO\_WAIT\_ON\_LOCK

In a terminal, this parameter is always forced to be `NO_WAIT_ON_LOCK`.

**offset** Byte offset of region to lock.

#### **offsetReference**

Interpretation of offset field. The parameter must be one of the following:

- FROM\_START\_OF\_FILE
- FROM\_CURRENT\_FILE\_POINTER
- FROM\_END\_OF\_FILE

**length** Length of region to lock in bytes.

This method throws a `FlexosException` exception if a file error occurs.

***read(byte[], int, int, int, int):*** Use this method to read data from the indicated record in a file. The file pointer is updated after every read to point to the byte following the last byte read. Before the read, the file pointer is positioned to the offset value referenced from the `offsetReference` reference. The method returns the number of bytes read into the buffer or a value of -1 if there is no data to read because the end of the file has been reached.

```
public int read(byte buffer[],
               int offset,
               int offsetReference,
               int readMethod,
               int length) throws FlexosException
```

#### **Parameters:**

**buffer** Buffer into which the data is read.

**offset** Byte offset to begin reading.

#### **offsetReference**

Interpretation of the offset field. The parameter must be one of the following:

- FROM\_START\_OF\_FILE
- FROM\_CURRENT\_FILE\_POINTER
- FROM\_END\_OF\_FILE

#### **readMethod**

Indicates whether to read from disk or read from cache. The parameter must be one of the following:

- READ\_FROM\_DISK
- READ\_FROM\_CACHE

**length** Number of bytes to read.

The method throws a `FlexosException` exception if the record cannot be read from the file.

***seek(int):*** This method sets the file pointer to the specified offset from the beginning of this file. The next read or write occurs at the new file pointer.

```
public void seek(int offset) throws FlexosException
```

**Parameters:**

**offset** Absolute position in bytes from the beginning of the file.

This method throws a `FlexosException` exception if a file error occurs.

***skipBytes(int):*** This method skips exactly *n* bytes of data and moves the file pointer *n* bytes. The method returns the number of bytes skipped, which is always *count*.

```
public void skipBytes(int count) throws FlexosException
```

**Parameters:**

**count** Number of bytes to be skipped.

This method throws a `FlexosException` exception if a file error occurs.

***unlock(int, int, int):*** Access to file records by other `POSFile` objects is restricted by locking and unlocking file records. Lock a record to restrict it; unlock the record to allow access. Typically, a record is locked before it is read and then unlocked after it is written back to the file. This method unlocks a record. The location of the record to be unlocked is determined by the file pointer. The file pointer is positioned to the offset value referenced from the `offsetReference` reference.

```
public void unlock(int offset,
                  int offsetReference,
                  int length) throws FlexosException
```

**Parameters:**

**offset** Byte offset of region to unlock.

**offsetReference**

Interpretation of `offset` field. The parameter must be one of the following:

- `FROM_START_OF_FILE`
- `FROM_CURRENT_FILE_POINTER`
- `FROM_END_OF_FILE`

**length** Length of region to unlock in bytes.

This method throws a `FlexosException` exception if a file error occurs.

***write(byte[], int, int, short, int):*** This method writes data into the indicated record. The file pointer is updated after every write to point to the byte following the last byte written. Before the write, the file pointer is positioned to the offset value referenced from the `offsetReference` reference.

```
public void write(byte buffer[],
                 int offset,
                 int offsetReference,
                 short flush,
                 int length) throws FlexosException
```

**Parameters:**

**buffer** Buffer containing the data to be written.

**offset** Byte offset to begin writing.

**offsetReference**

Interpretation of `offset` field. The parameter must be one of the following:

- FROM\_START\_OF\_FILE
- FROM\_CURRENT\_FILE\_POINTER
- FROM\_END\_OF\_FILE

**flush** Indicates whether to force data to the disk. This parameter forces the data to the disk. The parameter must be one of the following:

- FLUSH
- NO\_FLUSH

**length** Number of bytes to write.

The method throws a `FlexosException` exception if data cannot be written to the file.

***writeAppend(byte[], int):*** This method writes a record at the end of the file. The file pointer is updated after every write to point to the byte following the last byte written.

```
public void writeAppend(byte buffer[],
                        int length) throws FlexosException
```

#### Parameters:

**buffer** Buffer containing the data to be written.

**length** Number of bytes to write.

The method throws a `FlexosException` exception if data cannot be written to the file.

## Class KeyedFile

This section contains information on keyed files. Keyed files contain fixed length records that are accessed by using a key that is hashed to determine the location of a record in a file. The operating system provides a `KeyedFile` class that contains methods that can be used to provide reading and writing restrictions for other instances of this class that represent the same file. The constructors that create a keyed file cannot be used in a terminal.

```
public class com.ibm.OS4690.KeyedFile
    extends Object
    implements Def4690
{
    //Fields
    public static final byte COMPOUND_FILE
    public static final byte DISTRIBUTE_ON_CLOSE
    public static final byte DISTRIBUTE_ON_UPDATE
    public static final int EXCLUSIVE_ACCESS
    public static final int HOLD
    public static final int LOCK
    public static final byte MIRRORED_FILE
    public static final int NO_HOLD
    public static final int NO_LOCK
    public static final int NO_UNLOCK
    public static final int SHARED_READ_ACCESS
    public static final int SHARED_READ_WRITE_ACCESS
    public static final int UNLOCK

    //Constructors
    public KeyedFile(File, String, int)
    public KeyedFile(File, String, int, byte, byte, int, int, int, int, int)
    public KeyedFile(File, String, int, int, int, int, int, int)
    public KeyedFile(String, String, int)
    public KeyedFile(String, String, int, byte, byte, int, int, int, int, int)
    public KeyedFile(String, String, int, int, int, int, int, int)

    //Methods
    public void closeFull()
```

```

    public void closePartial()
    public void delete(byte[])
    protected void finalize()
    public int getKeyLength()
    public int getRecordSize()
    public int read(byte[], int)
    public int write(byte[], int, int)
}

```

## KeyedFile Fields

This section explains the fields for the KeyedFile class.

### ***COMPOUND\_FILE:***

```
public static final byte COMPOUND_FILE
```

This field indicates a file type. A read-only image of the file on the file server resides on all other controllers.

### ***DISTRIBUTE\_ON\_CLOSE:***

```
public static final byte DISTRIBUTE_ON_CLOSE
```

This field is a distribution indication; distribute the file when the file is closed.

### ***DISTRIBUTE\_ON\_UPDATE:***

```
public static final byte DISTRIBUTE_ON_UPDATE
```

This field is a distribution indication; distribute the file when the file is updated.

### ***EXCLUSIVE\_ACCESS:***

```
public static final int EXCLUSIVE_ACCESS
```

This field indicates an access restriction. Only this object is allowed read and write access.

### ***HOLD:***

```
public static final int HOLD
```

This field indicates to hold the write data.

### ***LOCK:***

```
public static final int LOCK
```

This field indicates to lock the record before a read.

### ***MIRRORED\_FILE:***

```
public static final byte MIRRORED_FILE
```

This field indicates a file type. A read-only image of the file on the file server resides on the alternate file server.

### ***NO\_HOLD:***

```
public static final int NO_HOLD
```

This field indicates not to hold the write data.

**NO\_LOCK:**

```
public static final int NO_LOCK
```

This field indicates not to lock the record before the read.

**NO\_UNLOCK:**

```
public static final int NO_UNLOCK
```

This field specifies not to unlock the record after a write.

**SHARED\_READ\_ACCESS:**

```
public static final int SHARED_READ_ACCESS
```

This field indicates an access restriction. Only this object is allowed write access to the file. Any object can have read access.

**SHARED\_READ\_WRITE\_ACCESS:**

```
public static final int SHARED_READ_WRITE_ACCESS
```

This field indicates an access restriction. Any object can access the file.

**UNLOCK:**

```
public static final int UNLOCK
```

This field specifies to unlock the record after a write.

**KeyedFile Constructors**

This section describes the constructors for the KeyedFile class.

**KeyedFile(File, String, int):** This constructor opens an existing keyed file to read from and, optionally, write to the file specified by the *file* parameter.

```
public KeyedFile(File file,
                 String mode,
                 int access) throws FlexosException, InvalidParameterException
```

**Parameters:**

**file**     The file object.

**mode**    Indicates either to open the file for input or for both input and output. The parameter must be either:

**r**        Read

**rw**       Read/Write

**access**

Indicates the file access restrictions. The parameter must be one of the following:

- EXCLUSIVE\_ACCESS
- SHARED\_READ\_ACCESS
- SHARED\_READ\_WRITE\_ACCESS

This constructor throws a `FlexosException` exception if the file cannot be opened. It throws an `InvalidParameterException` exception if an incorrect parameter is passed.

**KeyedFile(File, String, int, byte, byte, int, int, int, int, int):** This constructor creates and opens a distributed keyed file to read from and write to the file specified by the *file* parameter. Keyed files cannot be created by a class running in a terminal.



```

public KeyedFile(File file,
                 String mode,
                 int access,
                 byte fileType,
                 byte distributionMethod,
                 int keyLength,
                 int recordSize,
                 int numberOfRecords,
                 int randomizingDivisor,
                 int chainingThreshold) throws FlexosException,
                                     InvalidParameterException

```

### Parameters:

**file** The file object.

**mode** Indicates to open the file for both input and output. The parameter must be:

**rw** Read/Write

### **access**

Indicates the file access restrictions. The parameter must be one of the following:

- EXCLUSIVE\_ACCESS
- SHARED\_READ\_ACCESS
- SHARED\_READ\_WRITE\_ACCESS

### **fileType**

Indicates the file distribution type. The parameter must be one of the following:

- MIRRORED\_FILE
- COMPOUND\_FILE

### **distributionMethod**

Indicates when the file is to be distributed. The parameter must be one of the following:

- DISTRIBUTE\_ON\_UPDATE
- DISTRIBUTE\_ON\_CLOSE

### **keyLength**

Specifies the size of the key that is used to access records in the file. The key can be any length. The keyLength parameter plus the length of the data contained in the record equals the recordSize parameter.

### **recordSize**

Specifies the size of each record in the file. The maximum record length is 508 bytes. A record consists of the key and the data being written to the file.

### **numberOfRecords**

Specifies the number of records anticipated to be in the file. To allow for adequate record chaining, this number should be 25% greater than the maximum number of records expected.

### **randomizingDivisor**

Minimizes the number of chained records in a keyed file. This parameter must be less than the total blocks in the file. An example of a good randomizingDivisor could be the highest prime number that is less than the number of blocks in the keyed file. The number of blocks is determined by multiplying the numberOfRecords parameter by the recordSize parameter and dividing the result by 508.

### **chainingThreshold**

Number of sectors that are searched to find a keyed record before a message (W768) is logged. This must be a value of 1 to 255. The application is not notified that the threshold has been exceeded.

This constructor throws a `FlexosException` exception if the file already exists or cannot be created. It throws an `InvalidParameterException` exception if an incorrect parameter is passed.

***KeyedFile(File, String, int, int, int, int, int, int)***: This constructor creates and opens a local keyed file to read from and write to the file specified by the *file* parameter. Keyed files cannot be created by a class running in a terminal.

```
public KeyedFile(File file,
                 String mode,
                 int access,
                 int keyLength,
                 int recordSize,
                 int numberOfRecords,
                 int randomizingDivisor,
                 int chainingThreshold) throws FlexosException,
                                     InvalidParameterException
```

#### Parameters:

**file**     The file object.

**mode**    Indicates to open the file for both input and output. The parameter must be:

**rw**     Read/Write

#### **access**

Indicates the file access restrictions. The parameter must be one of the following:

- `EXCLUSIVE_ACCESS`
- `SHARED_READ_ACCESS`
- `SHARED_READ_WRITE_ACCESS`

#### **keyLength**

Specifies the size of the key that is used to access records in the file. The key can be any length. The `keyLength` parameter plus the length of the data contained in the record equals the `recordSize` parameter.

#### **recordSize**

Specifies the size of each record in the file. The maximum record length is 508 bytes. A record consists of the key and the data being written to the file.

#### **numberOfRecords**

Specifies the number of records anticipated to be in the file. To allow for adequate record chaining, this number should be 25% greater than the maximum number of records expected.

#### **randomizingDivisor**

Minimizes the number of chained records in a keyed file. This parameter must be less than the total blocks in the file. An example of a good `randomizingDivisor` could be the highest prime number that is less than the number of blocks in the keyed file. The number of blocks is determined by multiplying the `numberOfRecords` parameter by the `recordSize` parameter and dividing the result by 508.

#### **chainingThreshold**

Number of sectors that are searched to find a keyed record before a message (W768) is logged. This must be a value of 1 to 255. The application is not notified that the threshold has been exceeded.

This constructor throws a `FlexosException` exception if the file already exists or cannot be created. It throws an `InvalidParameterException` exception if an incorrect parameter is passed.

***KeyedFile(String, String, int)***: This constructor opens an existing keyed file to read from and, optionally, write to the file with the specified name.

```
public KeyedFile(String name,
                 String mode,
                 int access) throws FlexosException, InvalidParameterException
```

#### Parameters:

**name** File name.

**mode** Indicates either to open the file for input or for both input and output. The parameter must be either:

**r** Read

**rw** Read/Write

#### access

Indicates the file access restrictions. The parameter must be one of the following:

- EXCLUSIVE\_ACCESS
- SHARED\_READ\_ACCESS
- SHARED\_READ\_WRITE\_ACCESS

This constructor throws a `FlexosException` exception if the file cannot be opened. It throws an `InvalidParameterException` exception if an incorrect parameter is passed.

***KeyedFile(String, String, int, byte, byte, int, int, int, int, int):*** This constructor creates and opens a distributed keyed file to read from and write to the file with the specified name. Keyed files cannot be created by a class running in a terminal.

```
public KeyedFile(String name,
                 String mode,
                 int access,
                 byte fileType,
                 byte distributionMethod,
                 int keyLength,
                 int recordSize,
                 int numberOfRecords,
                 int randomizingDivisor,
                 int chainingThreshold) throws FlexosException,
                                           InvalidParameterException
```

#### Parameters:

**name** File name. If the file is not to be created in the current directory, *name* must include the path specification. The maximum length is 128 characters.

**mode** Indicates to open the file for both input and output. The parameter must be:

**rw** Read/Write

#### access

Indicates the file access restrictions. The parameter must be one of the following:

- EXCLUSIVE\_ACCESS
- SHARED\_READ\_ACCESS
- SHARED\_READ\_WRITE\_ACCESS

#### fileType

Indicates the file distribution type. The parameter must be one of the following:

- MIRRORED\_FILE
- COMPOUND\_FILE

#### distributionMethod

Indicates when the file is to be distributed. The parameter must be one of the following:

- `DISTRIBUTE_ON_UPDATE`
- `DISTRIBUTE_ON_CLOSE`

#### **keyLength**

Specifies the size of the key that is used to access records in the file. The key can be any length. The `keyLength` parameter plus the length of the data contained in the record equals the `recordSize` parameter.

#### **recordSize**

Specifies the size of each record in the file. The maximum record length is 508 bytes. A record consists of the key and the data being written to the file.

#### **numberOfRecords**

Specifies the number of records anticipated to be in the file. To allow for adequate record chaining, this number should be 25% greater than the maximum number of records expected.

#### **randomizingDivisor**

Minimizes the number of chained records in a keyed file. This parameter must be less than the total blocks in the file. An example of a good `randomizingDivisor` could be the highest prime number that is less than the number of blocks in the keyed file. The number of blocks is determined by multiplying the `numberOfRecords` parameter by the `recordSize` parameter and dividing the result by 508.

#### **chainingThreshold**

Number of sectors that are searched to find a keyed record before a message (W768) is logged. This must be a value of 1 to 255. The application is not notified that the threshold has been exceeded.

This constructor throws a `FlexosException` exception if the file already exists or cannot be created. It throws an `InvalidParameterException` exception if an incorrect parameter is passed.

***KeyedFile(String, String, int, int, int, int, int, int):*** This constructor creates and opens a local keyed file to read from and write to the file with a specified name. Keyed files cannot be created by a class running in a terminal.

```
public KeyedFile(String name,
                 String mode,
                 int access,
                 int keyLength,
                 int recordSize,
                 int numberOfRecords,
                 int randomizingDivisor,
                 int chainingThreshold) throws FlexosException,
                                     InvalidParameterException
```

#### **Parameters:**

**name** File name. If the file is not to be created in the current directory, *name* must include the path specification. The maximum length is 128 characters.

**mode** Indicates to open the file for both input and output. The parameter must be:

**rw** Read/Write

#### **access**

Indicates the file access restrictions. The parameter must be one of the following:

- `EXCLUSIVE_ACCESS`
- `SHARED_READ_ACCESS`
- `SHARED_READ_WRITE_ACCESS`

**keyLength**

Specifies the size of the key that is used to access records in the file. The key can be any length. The `keyLength` parameter plus the length of the data contained in the record equals the `recordSize` parameter.

**recordSize**

Specifies the size of each record in the file. The maximum record length is 508 bytes. A record consists of the key and the data being written to the file.

**numberOfRecords**

Specifies the number of records anticipated to be in the file. To allow for adequate record chaining, this number should be 25% greater than the maximum number of records expected.

**randomizingDivisor**

Minimizes the number of chained records in a keyed file. This parameter must be less than the total blocks in the file. An example of a good `randomizingDivisor` could be the highest prime number that is less than the number of blocks in the keyed file. The number of blocks is determined by multiplying the `numberOfRecords` parameter by the `recordSize` parameter and dividing the result by 508.

**chainingThreshold**

Number of sectors that are searched to find a keyed record before a message (W768) is logged. This must be a value of 1 to 255. The application is not notified that the threshold has been exceeded.

This constructor throws a `FlexosException` exception if the file already exists or cannot be created. It throws an `InvalidParameterException` exception if an incorrect parameter is passed.

**KeyedFile Methods**

This section describes the methods for the `KeyedFile` class.

***closeFull():*** Use this method to close this file stream and release any system resources associated with the stream.

```
public void closeFull() throws FlexosException
```

This method throws a `FlexosException` exception if a file error occurs.

***closePartial():*** This method is a partial close that flushes the buffers, but leaves the file open.

```
public void closePartial() throws FlexosException
```

The method throws a `FlexosException` exception if a file error occurs.

***delete(byte[]):*** Use this method to delete a record in the file. The key must be specified at the front of the buffer. `buffer[0]` is the most significant byte of the key.

```
public void delete(byte buffer[]) throws FlexosException, InvalidParameterException
```

**Parameters:**

**buffer** Buffer that contains the record to be deleted.

The method throws a `FlexosException` exception if the record cannot be deleted. It throws an `InvalidParameterException` exception if an incorrect parameter is passed.

***finalize():*** This method ensures that a full close is done for this file stream when it is no longer referenced.

```
protected void finalize()
```

The method overrides `finalize` in the `Object` class.

**getKeyLength():** Use this method to get the length of the key. The method returns the length of the key associated with this file.

```
public int getKeyLength() throws FlexosException
```

**getRecordSize():** Use this method to get the size of a file record. This method returns the size of a record in this file.

```
public int getRecordSize() throws FlexosException
```

**read(byte[], int):** Use this method to read a keyed record into a data buffer. The key must be specified at the front of the buffer. buffer[0] is the most significant byte of the key. The record data is added to the buffer after the key. The buffer must be large enough to contain a complete record. The method returns the number of bytes read into the buffer.

```
public int read(byte buffer[],
               int lockRecord) throws FlexosException, InvalidParameterException
```

#### Parameters:

**buffer** Buffer that contains the complete record.

#### lockRecord

Specifies whether the record is locked before it is read. The parameter must be one of the following:

- LOCK
- NO\_LOCK

The method throws a `FlexosException` exception if the record cannot be read from the file. It throws an `InvalidParameterException` exception if an incorrect parameter is passed.

**write(byte[], int, int):** This method writes a keyed record from a data buffer. The key must be specified at the front of the buffer. buffer[0] is the most significant byte of the key. The record can be locked before it is read and unlocked after the record is written. This method returns the number of bytes written.

```
public int write(byte buffer[],
               int unlockRecord,
               int hold) throws FlexosException, InvalidParameterException
```

#### Parameters:

**buffer** Buffer that contains the complete record.

#### unlockRecord

Specifies whether the record is unlocked after the record is written. The parameter must be one of the following:

- UNLOCK
- NO\_UNLOCK

**hold** Specifies whether to prevent the data from being physically written to the disk until the next write with hold is issued by this object. The parameter must be one of the following:

- HOLD
- NO\_HOLD

The method throws a `FlexosException` exception if the record cannot be written to the file. It throws an `InvalidParameterException` exception if an incorrect parameter is passed.

## Class POSPipeInputStream

The operating system provides a `POSPipeInputStream` class that you can use to read data from a POS pipe. A POS pipe can be either a local or remote pipe.

```

public class com.ibm.OS4690.POSPipeInputStream
    extends InputStream
    implements Def4690
{
    //Constructors
    public POSPipeInputStream(String, int)
    public POSPipeInputStream(int, char)

    //Methods
    public int available()
    public void close()
    protected void finalize()
    public int read()
    public int read(byte[], int, int)
    public int read(byte[])
    public long skip(long)
}

```

## POSPipeInputStream Constructors

This section describes the constructors for the `POSPipeInputStream` class.

***POSPipeInputStream(string, int):*** This constructor creates an input file stream to read from a local pipe with the specified name. A local pipe is used to receive data from another application or thread in the same terminal or controller. The *name* must begin with *pi:* and have the form *pi:pipeName*, with a maximum of 11 characters. The maximum size of a local pipe is 65 536 bytes.

```

public POSPipeInputStream(string name, int pipeSize)
    throws FlexosException, InvalidParameterException

```

### Parameters:

**name** Name of the pipe that is created.

### pipeSize

Size of the pipe (in bytes) that is created.

This constructor throws a `FlexosException` exception, if the pipe cannot be created. It throws an `InvalidParameter` exception, if an incorrect parameter is passed.

**Note:** `POSPipeInputStream` constructors attempt to create the pipe. If the pipe has already been created, then using the `POSPipeInputStream` constructors results in a `FlexosException` being thrown. If a Java program must read from an already-created pipe, the Java program can use the `POSFile` interface. For example:

```

POSFile pipe = new POSFile ("pi:" + pipeName, "r", POSFile.SHARED_READ_ACCESS);
byte[] pipeContents;
if (pipe.read(pipeContents, 0,
    POSFile.FROM_START_OF_FILE, POSFile.READ_FROM_DISK, recordSize) != -1)
{
    // pipeContents contains a record from the pipe
}
// pipeContents contains all pipe records available at the time of the reads

```

***POSPipeInputStream(int, char):*** This constructor creates an input file stream to read from a remote pipe with the specified ID. A remote pipe is used to receive data from another application or thread in a different terminal or controller. The maximum size of a remote terminal pipe is 240 bytes. If a larger size is specified, the pipe size is 240 bytes. No error is indicated.

The maximum size of a controller pipe is 65 536 bytes. If a larger size is specified, the pipe size is 65 536 bytes. No error is indicated.

```

public POSPipeInputStream(int pipeSize,
    char pipeID) throws FlexosException,
    InvalidParameterException

```

## Parameters:

### **pipeSize**

Size of pipe that is created.

### **pipeID**

One character (A to Z) that identifies the pipe.

This constructor throws a `FlexosException` exception, if the pipe cannot be created. It throws an `InvalidParameterException` exception, if an incorrect parameter is passed.

## POSPipeInputStream Methods

This section describes the methods for the `POSPipeInputStream` class.

**available():** Use this method in a controller to return the number of bytes that can be read from this input stream without blocking. If this method is run in a terminal, a return value greater than zero indicates there are at least that many bytes available to read.

```
public int available() throws FlexosException
```

This method throws a `FlexosException` exception, if there is an operating system error. This method overrides `available` in the `InputStream` class.

**close():** This method closes this input stream and releases any system resources associated with the stream.

```
public void close() throws FlexosException
```

This method throws a `FlexosException` exception if it cannot access the pipe. The method overrides `close` in the `InputStream` class.

**finalize():** This method ensures that a close is done for this file stream when it is no longer referenced.

```
protected void finalize()
```

The method overrides `finalize` in the `Object` class.

**read():** Use this method to read the next byte of data from this input stream. The value is returned as an integer in the range of 0 to 255. This method blocks until input data is available or an exception is thrown.

```
public int read() throws FlexosException
```

The method throws a `FlexosException` exception, if it cannot read the pipe. The method overrides `read` in the `InputStream` class.

**read(byte[]):** Use this method to read a number of bytes specified by the buffer size from this input stream into an array of bytes. This method blocks on a local pipe until all the input data is available or an exception is thrown. If this method is reading a remote terminal pipe, the maximum read size is 120 bytes. The maximum read size for all other pipes is 65 536 bytes.

```
public int read(byte buffer[]) throws FlexosException, InvalidParameterException
```

## Parameters:

**buffer** Buffer into which the data is read.

The method returns the number of bytes read from the pipe.

The method throws a `FlexosException` exception if the pipe cannot be read. It throws an `InvalidParameterException` exception if an incorrect parameter is passed. The method overrides `read` in the `InputStream` class.



***read(byte[], int, int):*** Use this method to read a specified number of bytes of data from this input stream into an array of bytes. This method blocks on a local pipe read until all the input data is available or an exception is thrown. If this method is reading a remote terminal pipe, the maximum read size is 120 bytes. The maximum read size for all other pipes is 65 536 bytes.

```
public int read(byte buffer[],
               int offset,
               int length) throws FlexosException, InvalidParameterException
```

#### Parameters:

**buffer** Buffer into which the data is read.

**offset** Area where data is located in buffer.

**length** Number of bytes to read.

This method returns the number of bytes read from the pipe.

The method throws a `FlexosException` exception if the pipe cannot be read. It throws an `InvalidParameterException` exception if an incorrect parameter is passed. The method overrides `read` in the `InputStream` class.

***skip(long):*** Use this method to skip and discard from this input stream the number of bytes specified by *count*. The method can skip a smaller number of bytes. The actual number of bytes skipped is returned.

```
public long skip(long count) throws FlexosException, InvalidParameterException
```

#### Parameters:

**count** Number of bytes to skip

This method throws a `FlexosException` exception if it cannot access the pipe. It throws an `InvalidParameterException` exception if an incorrect parameter is passed. The method overrides `skip` in the `InputStream` class.

## Class `POSPipeOutputStream`

The operating system provides a `POSPipeOutputStream` class that you can use to write data to a POS pipe. A POS pipe can be either a local or remote pipe.

```
public class com.ibm.OS4690.POSPipeOutputStream
    extends OutputStream
    implements Def4690
{
    //Constructors
    public POSPipeOutputStream(String)
    public POSPipeOutputStream(char, String)

    //Methods
    public void close()
    protected void finalize()
    public void write(byte[])
    public void write(byte[], int, int)
    public void write(int)
    public void writeConditional(byte[])
    public void writeConditional(byte[], int, int)
    public void writeConditional(int)
}
```

## `POSPipeOutputStream` Constructors

This section describes constructors for the `POSPipeOutputStream` class.

**POSPipeOutputStream(string):** This constructor opens an output file stream to write to a local pipe with the specified name. A local pipe is used to receive data from another application or thread in the same terminal or controller. The *name* must begin with *pi:* and must have the form *pi:pipename*, with a maximum of 11 characters.

```
public POSPipeOutputStream(string name)
    throws FlexosException, InvalidParameterException
```

**Parameters:**

**name** Name of pipe to be opened.

This constructor throws a *FlexosException* exception if the pipe cannot be opened. It throws an *InvalidParameterException* exception if an incorrect parameter is passed.

**POSPipeOutputStream(char, string):** This constructor opens an output file stream to write to a remote pipe with the specified pipe ID. A remote pipe is used to send data to another application or thread in a different terminal or controller. The maximum amount of data that can be written to a remote terminal pipe is 120 bytes.

```
public POSPipeOutputStream(char pipeID, String node)
    throws FlexosException, InvalidParameterException
```

**Parameters:**

**pipeID**

One character (A to Z) that identifies the pipe.

**node** Specifies the terminal number (001 to 999) or controller node (*xy*), where *xy* is C-Z, AA, or BB.

This constructor throws a *FlexosException* exception if the pipe cannot be opened. It throws an *InvalidParameterException* exception if an incorrect parameter is passed.

## POSPipeOutputStream Methods

This section describes the methods for the *POSPipeOutputStream* class.

**close():** This method closes this output stream and releases any system resources associated with the stream.

```
public void close() throws FlexosException
```

The method throws a *FlexosException* exception if a pipe error occurs. The method overrides *close* in the *OutputStream* class.

**finalize():** This method ensures that a close is done for this pipe when it is no longer referenced.

```
protected void finalize()
```

The method overrides *finalize* in the *Object* class.

**write(byte[]):** Use this method to write a number of bytes specified by the buffer size to this output stream from an array of bytes. This method blocks until all the data is written to the pipe or an exception is thrown. For a remote terminal pipe, the buffer length cannot exceed 120 bytes.

```
public void write(byte buffer[]) throws FlexosException
```

**Parameters:**

**buffer** Buffer containing data to be written.

This method throws a *FlexosException* exception if it cannot write to the pipe. The method overrides *write* in the *POSPipeOutputStream* class.

***write(byte[], int, int):*** Use this method to write x number of bytes from the specified byte array starting at the offset parameter to this output stream. This method blocks until all the data is written to the pipe or an exception is thrown. For a remote terminal pipe, the buffer length cannot exceed 120 bytes.

```
public void write(byte buffer[],
                  int offset,
                  int length) throws FlexosException, InvalidParameterException
```

**Parameters:**

**buffer** Buffer containing data to be written.

**offset** offset into the buffer.

**length** Number of bytes to write.

This method throws a `FlexosException` exception if it cannot write to the pipe. It throws an `InvalidParameterException` exception if an incorrect parameter is passed. The method overrides `write` in the `POSPipeOutputStream` class.

***write(int):*** Use this method to write the specified byte of data to this output stream. This method blocks until the data is written to the pipe or an exception is thrown.

```
public void write(int data) throws FlexosException, InvalidParameterException
```

**Parameters:**

**data** Data to write to the output stream. This is an integer from 0 to 255.

This method throws a `FlexosException` exception if it cannot write to the pipe. It throws an `InvalidParameterException` exception if an incorrect parameter is passed. The method overrides `write` in the `POSPipeOutputStream` class.

***writeConditional(byte[]):*** Use this method to write a number of bytes specified by the buffer size to this output stream from an array of bytes. This method throws an exception with a return code of -1 immediately if the data cannot be written to the pipe. This method can only be used to write to remote pipes.

```
public void writeConditional(byte buffer[]) throws FlexosException
```

**Parameters:**

**buffer** Buffer containing data to be written.

This method throws a `FlexosException` exception with a return code of -1 if the pipe is full; it throws a `FlexosException` exception with a return code of 80104010 if the pipe does not exist.

***writeConditional(byte[], int, int):*** Use this method to write x bytes from the specified byte array starting at the offset parameter to this output stream. This method throws an exception with a return code of -1 immediately if the data cannot be written to the pipe. This method can only be used to write to remote pipes.

```
public void writeConditional(byte buffer[],
                             int offset,
                             int length) throws FlexosException,
                                                InvalidParameterException
```

**Parameters:**

**buffer** Buffer containing data to be written.

**offset** offset into the buffer.

**length** Number of bytes to write.

This method throws a FlexosException exception with a return code of -1 if the pipe is full; it throws a FlexosException exception with a return code of 80104010 if the pipe does not exist. It throws an InvalidParameterException exception if an incorrect parameter is passed.

**writeConditional(int):** Use this method to write the specified byte of data to this output stream. This method throws an exception with a return code of -1 immediately if the data cannot be written to the pipe. This method can only be used to write to remote pipes.

```
public void writeConditional(int data) throws FlexosException,
                                   InvalidParameterException
```

#### Parameters:

**data** Data to write to the output stream. This is an integer from 0 to 255.

This method throws a FlexosException exception with a return code of -1 if the pipe is full; it throws a FlexosException exception with a return code of 80104010 if the pipe does not exist. It throws an InvalidParameterException exception if an incorrect parameter is passed.

## Class ControllerApplicationServices

Store controller applications use a class called ControllerApplicationServices class to request Application Services (ADXSERVE) functions. ControllerApplicationServices class is used by Java to access these operating system services.

```
public class com.ibm.OS4690.ControllerApplicationServices
    extends Object
    implements Def4690
{
    //Methods
    public static void disableControllerProgrammablePower()
    public static void disableStorageRetention()
    public static void dumpSystemStorage()
    public static String[] getAllActiveNetworkControllers()
    public static String[] getConfiguredNetworkControllers()
    public static String getControllerId(short)
    public static byte[] getControllerModelAndType()
    public static ControllerStatusData getControllerStatus()
    public static int getLoopIDForTerminal(short)
    public static String[] getLoopMessage(String, byte)
    public static byte[] getLoopStatus()
    public static boolean isController()
    public static String getStoreControllerIDForTerminal(short)
    public static boolean isDumpFlagOn()
    public static boolean isEntriesInFileServerExceptionLog()
    public static boolean isEntriesInMasterExceptionLog()
    public static void logError(char, int, int, int, byte[])
    public static void powerOffLocalMachine(String)
    public static void powerOffRemoteController(String)
    public static void powerOffRemoteTerminal(String)
    public static void resetDumpFlag()
    public static void setDumpFlag()
    public static void setLocalControllerProgrammablePower()
    public static void setStorageRetention()
    public static void setTerminalSleepModeTimer(short)
    public static void startBackgroundApplication(String, byte[], String, short)
    public static int runRSTConTerm(int)
    public static void runRSTConAllTerms()
    public static void ip1Controller()
    public static void ip1AllControllers()
    public static int startBackgroundApplicationSameSlot(String)
    public static int stopBackgroundApplicationSameSlot(String)
    public static void logSystemMessage(int, int, int, char, int, int, byte[])
    public static void waitForKeyboardMouseActivity()
```

```

    public static void waitForKeyboardMouseInactivity(short waitTime)
    public static void signOffActiveUser()
    public static void disconnectActiveUser()
}

```

## ControllerApplicationServices methods

This section describes the methods for the ControllerApplicationServices class.

***disableControllerProgrammablePower():*** Programmable power enables controller applications to use methods to enable or disable the programmable power feature and to use methods to power off a terminal or controller. This method disables programmable power on the local controller. This method must be invoked on the controller on which programmable power is to be disabled. The controller that invokes this method must be in the 469 x or SurePOS 700 family to have the programmable power feature.

```
public static void disableControllerProgrammablePower() throws FlexosException
```

The following FlexosException exception return codes are thrown by this method:

- 1015 A power-off message cannot be sent to the remote controller because the system is a non-LAN system.
- 1016 The machine to be powered off is not in the 4690 family or the machine is a controller/terminal environment.
- 1017 A date or time that is not valid was specified.
- 1018 The controller ID is not valid or is not the active controller.
- 1020 BIOS driver open failed.
- 1023 The programmable power request is pending. Either programmable power has been disabled or an application has set the NO-IPL flag.

***disableStorageRetention():*** This method disables storage retention in all of the terminals on the TCC Network of the store controller requesting the disable.

```
public static void disableStorageRetention() throws FlexosException
```

The method throws a FlexosException exception with a return code of -1080 if the command is blocked by a system command function from the screen and keyboard.

***dumpSystemStorage():*** This method causes all of the storage in the controller to be dumped to a disk file named ADXCSLCF.DAT in the root directory.

```
public static void dumpSystemStorage() throws FlexosException
```

***getAllActiveNetworkControllers():*** This method returns the IDs of all active store controllers on the network. Each ID is two bytes long. The number of elements in the array indicates the number of active controllers.

```
public static String[] getAllActiveNetworkControllers() throws FlexosException
```

***getConfiguredNetworkControllers():*** This method returns the IDs for all of the store controllers on the network. Each ID is two bytes long. Store controller IDs range from CC through ZZ. If there are less than 20 controllers, 00 (ASCII zeros, not numeric) indicates the end of the list. The number of elements in the array indicates the number of configured controllers.

```
public static String[] getConfiguredNetworkCongrollers() throws FlexosException
```

***getControllerId(short):*** This method returns the store controller ID for the specified terminal. The ID is returned as a two-character string showing the store controller ID (CC through ZZ). If the terminal number is not defined, the string contains 00. The returned string is valid only if this method is invoked at the master store controller.

```
public static String getControllerId(short termNum) throws FlexosException
```

**Parameters:****termNum**

Number of the terminal for which the ID is requested.

***getControllerModelAndType():*** This method returns the store controller model number and type. The machine model and type is returned as a byte representing the machine model and type. For example, for a 4693-541 machine, the values returned are X'F8'X'3E'X'00'X'00'

```
public static byte[] getControllerModelAndType() throws FlexosException
```

***getControllerStatus():*** This method returns an object of type `ControllerStatusData` has various methods to gather status information about the controller. Access to the status information is obtained by invoking the `ControllerStatusData` object's methods. Refer to "Class `ControllerStatusData`" on page 513 for information on the `ControllerStatusData` methods.

```
public static ControllerStatusData getControllerStatus() throws FlexosException
```

The method throws a `FlexosException` exception with a return code of 0x80004085 if you try to call `getControllerStatus` from a terminal application.

***getLoopIDForTerminal(short):***

**Note:** Store Loop support was removed in 4690 OS V6R3.

This method returns the two-character loop ID for a specified terminal. The returned string is valid only if this method is invoked at the master store controller.

```
public static int getLoopIDForTerminal(short terminalNumber) throws FlexosException
```

**Parameters:****terminalNumber**

Terminal Number.

**Returns** The possible return values are:

- 1      Store Loop Card 1
- 2      Store Loop Card 2
- 3      Controller/Terminal
- 4      LAN System (Ethernet or token ring)
- 5      RF System

The method throws a `FlexosException` exception with a return code of -1002 if the terminal number passed in cannot be found.

***getLoopMessage(String, byte):*** This method returns the three most recent TCC Network messages for the token ring adapter specified in the input buffer. Each message is 133 characters long, and the oldest message is returned first.

```
public static String[] getLoopMessage(String node, byte TCCNetwork)
                           throws FlexosException
```

**Parameters:**

**node**    Two-character node (for example, CC).

**TCCNetwork**

Byte representing the TCC Network (for example, 1 or 2).





**logError(char, int, int, int, byte[]):** This method is used to log an error in the application error log. The message should be in file ADXCSOZF.DAT. The application name is automatically included in the log entry.

```
public static void logError(char msggrp,  
                           int msgnum,  
                           int severity,  
                           int event,  
                           byte[] unique)  
    throws FlexosException InvalidParameterException
```

#### Parameters:

##### msggrp

Message group character, which is unique for each product and should be in the range of J to S.

##### msgnum

Message number, if any. If the message number is zero, no message is displayed. This number is converted to three printable decimal digits.

##### severity

Number ranging from 1 to 5 that indicates the importance of the message. The most important is severity 1.

**event** Value assigned by the application that is used to indicate why the error is being logged. This number is converted to three printable decimal digits.

##### unique

Byte array to be included in the message. The maximum length is 10 bytes.

**powerOffLocalMachine(String):** This method is used to power off a remote LAN (MCF Network) controller. This method can be invoked on a controller from any supported family (personal computer, 469x, SurePOS 700 or TCxWave 6140 Series). The controller on which this method is invoked must be the master controller in a LAN system. The remote controller being powered off must be in the 469x or SurePOS 700 family to have the programmable power feature. The remote LAN controller is powered off immediately when this method is called. The time in the string provided by the caller as input is the power-on time.

```
public static void powerOffLocalMachine(String powerData) throws FlexosException
```

#### Parameters:

##### powerData

Power on time (DDHHMM), where:

**DD** Day of the month (01–31)

**HH** Hours (00–24)

**MM** Minutes (00–59)

The following FlexosException exception return codes are thrown by this method:

- 1016** The machine to be powered off is not in the 4690 family or the machine is a controller/terminal environment.
- 1017** A date or time that is not valid was specified.
- 1018** The controller ID is not valid or is not the active controller.
- 1020** BIOS driver open failed.
- 1023** The programmable power request is pending. Either programmable power has been disabled or an application has set the NO-IPL flag.

**powerOffRemoteController(String):** This method is used to power off a remote LAN (MCF Network) controller. This method can be invoked on a controller from any supported family (personal computer or



469x or SUREPOS 700). The controller on which this method is invoked must be the master controller in a LAN system. The remote controller being powered off must be in the 469x or SurePOS 700 family to have the programmable power feature. The remote LAN controller is be powered off immediately when this method is called. The time in the string provided by the caller as input is the power-on time

```
public static void powerOffRemoteController(String powerData) throws FlexosException
```

#### Parameters:

##### **powerData**

Power on time (DDHHMMCC), where:

- DD** Day of the month (01–31)
- HH** Hours (00–24)
- MM** Minutes (00–59)
- CC** Controller ID (CC through ZZ)

The following `FlexosException` exception return codes are thrown by this method:

- 1015** A power-off message cannot be sent to the remote controller because the system is a non-LAN system.
- 1016** The machine to be powered off is not in the 4690 family or the machine is a controller/terminal environment.
- 1017** A date or time that is not valid was specified.
- 1018** The controller ID is not valid or is not the active controller.
- 1020** BIOS driver open failed.
- 1023** The programmable power request is pending. Either programmable power has been disabled or an application has set the NO-IPL flag.

***powerOffRemoteTerminal(String):*** This method is used to power off a remote terminal. This method can be invoked on a controller from any supported family (personal computer, 469x or SurePOS 700). The controller on which this method is invoked must be the master controller if on a LAN (MCF Network) system, or from the stand-alone controller. The remote terminal being powered off must be in the 469x, SurePOS 300/700 or TCxWave family to have the programmable power feature. The remote terminal is powered down immediately when this method is called if storage retention is disabled, otherwise it will go to a suspend state. The time in the string provided by the caller as input is the power-on time.

```
public static void powerOffRemoteTerminal(String powerData) throws FlexosException
```

#### Parameters:

##### **powerData**

Power off time (DDHHMMTTT), where

- DD** Day of the month (01–31)
- HH** Hours (00–24)
- MM** Minutes (00–59)
- TTT** Terminal Number (001–999)

The following `FlexosException` exception return codes are thrown by this method:

- 1015** A power-off message cannot be sent to the remote controller because the system is a non-LAN system.
- 1016** The machine to be powered off is not in the 4690 family or the machine is a controller/terminal environment.

- 1017 A date or time that is not valid was specified.
- 1018 The controller ID is not valid or is not the active controller.
- 1020 BIOS driver open failed.
- 1023 The programmable power request is pending. Either programmable power has been disabled or an application has set the NO-IPL flag.

**resetDumpFlag():** This method is used to reset the terminal preservation flag. While the flag is reset, the data currently in the dump file is allowed to be overwritten with another terminal dump when the next terminal dump request is received.

public static void resetDumpFlag() throws FlexosException

**setDumpFlag():** This method is used to set the terminal preservation flag. While the flag is set, the terminal dump requests are rejected so that the dump currently in the terminal dump file is not overwritten.

public static void setDumpFlag() throws FlexosException

**setLocalControllerProgrammablePower():** This method enables programmable power on the local controller. This method must be invoked on the controller on which programmable power is to be enabled. The controller that invokes this method must be in the 469 x or SurePOS 700 family to have the programmable power feature.

public static void setLocalControllerProgrammablePower() throws FlexosException

The following FlexosException exception return codes are thrown by this method:

- 1016 The machine to be powered off is not in the 4690 family or the machine is a controller/terminal environment.
- 1017 A date or time that is not valid was specified.
- 1018 The controller ID is not valid or is not the active controller.
- 1020 BIOS driver open failed.
- 1023 The programmable power request is pending. Either programmable power has been disabled or an application has set the NO-IPL flag.

**setStorageRetention():** This method enables storage retention in all of the terminals on the TCC Network of the store controller requesting the enable.

public static void setStorageRetention() throws FlexosException

The method throws a FlexosException exception with a return code of -1080 if the command is blocked by a system command function from the screen and keyboard.

**setTerminalSleepModeTimer(short):** This method enables an application running in a store controller to set the Terminal Sleep Mode Inactive Timeout. When Terminal Storage Retention is enabled, setting this value determines how many minutes a terminal remains idle before going into a standby or suspend state. If storage retention is disabled this activity timer will cause the terminal to be powered down.

public static void setTerminalSleepModeTimer(short timeout) throws FlexosException

#### Parameters:

##### timeout

Terminal sleep mode. Valid values are 0 to 255. A value of 0 disables terminal sleep mode.

**startBackgroundApplication(String, byte[], String, short):** This method enables a Java application to start a C or CBASIC background application on the operating system. It also enables you to set a priority level for the background application. This function is only valid for store controller applications.

```
public static void startBackgroundApplication (String applicationName,
                                             byte parameters[],
                                             String message,
                                             short priority)
    throws FlexosException
```

**Parameters:**

**applicationName**

Name of the application to be started. This should not exceed 21 characters.

**parameters**

Parameters for the application. This should not exceed 46 bytes.

**message**

Message to be displayed on the background status screen. This should not exceed 46 characters.

**priority**

Priority level for the background application. This should range from 1 to 9.

The following `FlexosException` exception return codes are thrown by the method:

- 1001 Message is greater than 46 characters in length.
- 1170 Application name is greater than 21 characters.
- 1171 All background entries are in use.
- 1172 The maximum number of background applications are already active.
- 1175 Length of the parameters exceeds 46 bytes.
- 1177 Starting priority is not between 1 and 9.

***startBackgroundApplicationSameSlot(String):*** This method enables a Java application to start a C or CBASIC background application in the same background slot.

```
public static int startBackgroundApplicationSameSlot(
String applicationAndParmsName,
throws FlexosException
```

**Parameters:**

**applicationAndParmsName:**

String specifying the application name (1 to 24 characters) and optionally appended parameter string (0 to 45 characters). If there is a parameter string, the name must be padded with blanks so that parameter data starts on character 25 of the string.

The following `FlexosException` exception return codes are thrown by the method:

- 1138 Background application already running.
- 1170 Invalid or missing program name in applicationAndParmsName string.
- 1173 Memory range error in applicationAndParmsName string.
- 1175 Other applicationAndParmsName string error.

***stopBackgroundApplicationSameSlot(String):*** This method enables a Java application to stop a previously started C or CBASIC background application in the same background slot.

```
public static int stopBackgroundApplicationSameSlot(
String applicationAndParmsName,
throws FlexosException
```

## Parameters:

### **applicationAndParmsName:**

String specifying the application name (1 to 24 characters) and optionally appended parameter string (0 to 45 characters). If there is a parameter string, the name must be padded with blanks so that parameter data starts on character 25 of the string.

The following FlexosException exception return codes are thrown by the method:

- 1139 Background application already stopped.
- 1170 Invalid or missing program name in applicationAndParmsName string.
- 1173 Memory range error in applicationAndParmsName string.
- 1175 Other applicationAndParmsName string error.

**runRSTConTerm(int):** This method causes Remote Set Terminal Characteristics to run on the terminal whose terminal number is passed in the parameter.

**runRSTConAllTerm():** This method causes Remote Set Terminal Characteristics to run on all of the terminals that are attached to the controller. No parameters are passed in this method.

**waitForKeyboardMouseInactivity (short waitTime):** This method returns when no controller keyboard, terminal Java keyboard, or mouse activity has occurred on the system console for the time specified (*waitTime*). Refer to “Programming auto sign off” on page 5 for more information about using Auto Sign Off.

```
public static void waitForKeyboardMouseInactivity (short waitTime) throws FlexosException
```

## Parameters:

### **waitTime**

Number of seconds of keyboard and mouse inactivity to wait for. The allowable range is 15 to 3600 seconds.

The following FlexosException exception return codes are thrown by this method:

- 1101 The requestor is not a background application.
- 1301 A keyboard and mouse wait is already in progress.
- 1302 The number of second specified to wait for inactivity (*waitTime*) is out of the allowed range.

**waitForKeyboardMouseActivity():** This method returns at the first controller keyboard, terminal Java keyboard, or mouse activity on the system console. Refer to “Programming auto sign off” on page 5 for more information about using Auto Sign Off.

```
public static void waitForKeyboardMouseActivity() throws FlexosException
```

The following FlexosException exception return codes are thrown by this method:

- 1101 The requestor is not a background application.
- 1301 A keyboard and mouse wait is already in progress.

**signoffActiveUser():** This method signs off the system console active user. Refer to “Programming auto sign off” on page 5 for more information about using Auto Sign Off.

```
public static void signoffActiveUser() throws FlexosException
```

The following FlexosException exception return codes are thrown by this method:

- 1101 The requestor is not a background application.
- 1303 Request only valid when system is in controller or terminal Java mode.
- 1304 Request not valid when there is not a system console active user.

**disconnectActiveUser():** This method disconnects the system console active user. Refer to “Programming auto sign off” on page 5 for more information about using Auto Sign Off.

```
public static void disconnectActiveUser() throws FlexosException
```

The following FlexosException exception return codes are thrown by this method:

- 1101 The requestor is not a background application.
- 1303 Request only valid when system is in controller or terminal Java Mode.
- 1304 Request not valid when there is not a system console active user.

**mediaEject():** This method allows an application to programmatically eject CD/DVD media inserted in the controller CD/DVD drive.

**Note:** Using the *Eject* function while writable media is in use may lead to data loss or damage to the media.

In order to prevent data loss, ensure that a *write* operation is not in progress. Issue the **unlockp** command before attempting to eject the media.

```
public static void mediaEject() throws FlexosException
```

The following FlexosException exception return codes are thrown by this method:

- 0x805B9700 = A CD/DVD operation was in or is in progress preventing the eject.
- 0x805B9701 = The drive is locked, you must issue an **unlockp** command prior to attempting the eject.
- 0x805B9702 = This is a fatal error and it signals that the low CD/DVD device driver was unable to be opened.
- 0x805B9703 = The unit's door is closing, retry the **eject** command.
- 0x805B9704 = The CD/DVD Device Driver failed to acknowledge the command.
- 0x805B9707 = The eject was not performed due to a **format** or **chkdsk** command being in progress.
- 1020 = A resource on the P: drive is in use, preventing the **eject** command from completing. Usually this indicates that the working directory on the P: drive is set to something other than the root directory.
- 0 = Success

**public static short startInactivityMonitoring(short waitTime) throws FlexosException:** This method returns when:

- No controller or terminal Java keyboard or mouse activity has occurred on the system console for the time specified.
- No keyboard activity has occurred on any auxiliary console for the time specified.

Refer to “Programming auto sign off for system and auxiliary consoles” on page 6 for more information about using auto signoff for system and auxiliary consoles.

Supported on enhanced systems only.

This function uses the following parameters:

**waitTime**

= amount of seconds of keyboard and mouse inactivity to wait for, range 15-3600 seconds (15 seconds – 1 hour)

**RET**

= One of the following values:

0x8000 = System console has had no keyboard or mouse activity for the specified amount of time.

0x0001-0x0008 = specified auxiliary console has had no keyboard activity for the specified amount of time

Both the system console and a single auxiliary console can return at the same time. For example, a return value of 0x8003 would indicate that the system console and auxiliary console 3 have had inactivity for the specified amount of time. All consoles that are not specified in the return value will continue to be monitored for inactivity.

This method throws a `FlexosException` exception with the following return codes:

- 1101 = Requestor not a background application.
- 1301 = Keyboard and mouse wait already in progress.
- 1302 = Amount of seconds to wait for inactivity (waitTime) out of range.
- 1304 = No active user on any consoles (system or auxiliary).

***public static void stopInactivityMonitoring() throws FlexosException:*** This method returns when keyboard and mouse inactivity monitoring for all consoles (system and auxiliary) has been stopped.

Supported on enhanced systems only.

This method throws a `FlexosException` exception with the following return codes:

- 1101 = Requestor not a background application.
- 1301 = Keyboard and mouse wait already in progress.

***public static void signOffConsoleActiveUser(short console) throws FlexosException:*** This method signs off the active users from the requested console.

Supported on enhanced systems only.

This function uses the following parameters:

Console:

- 0 = System console
- 1-8 = Auxiliary console 1,2,3,4,5,6,7, or 8
- 1 = All consoles (system and auxiliary)

This method throws a `FlexosException` exception with the following return codes:

- 1101 = Requestor not a background application.
- 1302 = Invalid console requested.
- 1303 = Request not valid for system console when c/t is in terminal mode.
- 1304 = Request not valid when there is not an active user for this console.
- 1305 = Request not valid when there is no auxiliary console configured for this console.

***public static void disconnectConsoleActiveUser(short console) throws FlexosException:*** This method disconnects the active users from the requested console.

Supported on enhanced systems only.

This function uses the following parameters:

Console:

- 0 = System console
- 1-8 = Auxiliary console 1,2,3,4,5,6,7, or 8
- 1 = All consoles (system and auxiliary)

This method throws a FlexosException exception with the following return codes:

- 1101 = Requestor not a background application.
- 1302 = Invalid console requested.
- 1303 = Request not valid for system console when c/t is in terminal mode.
- 1304 = Request not valid when there is not an active user for this console.
- 1305 = Request not valid when there is no auxiliary console configured for this console.

## Class ControllerStatusData

The getControllerStatus method returns an object of type ControllerStatusData that has various methods to gather status information about the controller. Access to the status information is obtained by invoking the ControllerStatusData object's methods.

```
public class com.ibm.OS4690.ControllerStatusData
    extends Object
{
    //Fields
    public static final String ALT_MASTER_ALT_FILESERVER
    public static final String ALTERNATE_FILE_SERVER
    public static final String ALTERNATE_MASTER
    public static final char COLOR_DISPLAY
    public static final char COMMA_CONVENTION
    public static final char CONSOLE_0
    public static final char CONSOLE_1
    public static final char CONSOLE_2
    public static final char CONSOLE_3
    public static final char CONSOLE_4
    public static final char CONSOLE_5
    public static final char CONSOLE_6
    public static final char CONSOLE_7
    public static final char CONSOLE_8
    public static final String CONTROLLER_NOT_ON_LAN
    public static final String CONTROLLER_ON_LAN
    public static final char D_M_Y_FORMAT
    public static final char DAY_MONTH_YEAR_FORMAT
    public static final char ETHERNET
    public static final String FILE_SERVER
    public static final char HMS_TIME_FORMAT
    public static final char HOUR_MINUTE_SECOND_FORMAT
    public static final char LAN_NOT_INSTALLED
    public static final char M_D_Y_FORMAT
    public static final String MASTER
    public static final String MASTER_AND_FILE_SERVER
    public static final char MONOCHROME_DISPLAY
    public static final char MONTH_DAY_YEAR_FORMAT
    public static final char PERIOD_CONVENTION
    public static final char TOKENRING
    public static final char TWO_SIGNIFICANT_DIGITS
    public static final char UNKNOWN_DISPLAY
    public static final char ZERO_SIGNIFICANT_DIGITS

    //Methods
    public boolean isControllerOnActiveLan()
    public String getActingControllerType()
    public char getAssignedConsole()
    public String getControllerId()
    public char getDateFormat()
    public char getDisplayType()
    public char getLanConnectionType()
    public String getMasterControllerId()
    public char getMonetaryFormat()
    public char getNumberOfDigitsAfterDecimal()
    public String getNumberPrinterLinesPerPage()
    public String getNumberTerminalsConfigured()
}
```



```

    public String getStoreNumber()
    public char getTimeFormat()
    public char getPrinterAssociatedWithConsole()
}

```

## ControllerStatusData Fields

This section describes the ControllerStatusData fields.

### ***ALT\_MASTER\_ALT\_FILESERVER:***

```
public static final String ALT_MASTER_ALT_FILESERVER
```

This field indicates that the acting controller is the alternate master and the alternate file server.

### ***ALTERNATE\_FILE\_SERVER:***

```
public static final String ALTERNATE_FILE_SERVER
```

This field indicates that the acting controller is the alternate file server.

### ***ALTERNATE\_MASTER:***

```
public static final String ALTERNATE_MASTER
```

This field indicates that the acting controller is the alternate master.

### ***COLOR\_DISPLAY:***

```
public static final char COLOR_DISPLAY
```

This field indicates that the display is a color display.

### ***COMMA\_CONVENTION:***

```
public static final char COMMA_CONVENTION
```

This field indicates that the monetary format uses the comma convention.

### ***CONSOLE\_0:***

```
public static final char CONSOLE_0
```

This field indicates that the console you are using is Console 0.

### ***CONSOLE\_1:***

```
public static final char CONSOLE_1
```

This field indicates that the console you are using is Console 1 or identifies with which console the printer is associated.

### ***CONSOLE\_2:***

```
public static final char CONSOLE_2
```

This field indicates that the console you are using is Console 2 or identifies with which console the printer is associated.

### ***CONSOLE\_3:***

```
public static final char CONSOLE_3
```

This field indicates that the console you are using is Console 3 or identifies with which console the printer is associated.



**CONSOLE\_4:**

```
public static final char CONSOLE_4
```

This field indicates that the console you are using is Console 4 or identifies with which console the printer is associated.

**CONSOLE\_5:**

```
public static final char CONSOLE_5
```

This field indicates that the console you are using is Console 5 or identifies with which console the printer is associated.

**CONSOLE\_6:**

```
public static final char CONSOLE_6
```

This field indicates that the console you are using is Console 6 or identifies with which console the printer is associated.

**CONSOLE\_7:**

```
public static final char CONSOLE_7
```

This field indicates that the console you are using is Console 7 or identifies with which console the printer is associated.

**CONSOLE\_8:**

```
public static final char CONSOLE_8
```

This field indicates that the console you are using is Console 8 or identifies with which console the printer is associated.

**CONTROLLER\_NOT\_ON\_LAN:**

```
public static final String CONTROLLER_NOT_ON_LAN
```

This field indicates that the controller is not on the LAN.

**CONTROLLER\_ON\_LAN:**

```
public static final String CONTROLLER_ON_LAN
```

This field indicates that the controller is on the LAN.

**D\_M\_Y\_FORMAT:**

```
public static final char D_M_Y_FORMAT
```

This field indicates that the date format is day.month.year.

**DAY\_MONTH\_YEAR\_FORMAT:**

```
public static final char DAY_MONTH_YEAR_FORMAT
```

This field indicates that the date format is day/month/year.

**ETHERNET:**

```
public static final char ETHERNET
```

This field indicates that the LAN connection type is Ethernet.

**FILE\_SERVER:**

```
public static final String FILE_SERVER
```

This field indicates that the acting controller is the file server.

**HMS\_TIME\_FORMAT:**

```
public static final char HMS_TIME_FORMAT
```

This field indicates that the time format is hour.minute.second.

**HOURLY\_MINUTE\_SECOND\_FORMAT:**

```
public static final char HOURLY_MINUTE_SECOND_FORMAT
```

This field indicates that the time format is hour:minute:second.

**LAN\_NOT\_INSTALLED:**

```
public static final char LAN_NOT_INSTALLED
```

This field indicates that a LAN system is not installed.

**M\_D\_Y\_FORMAT:**

```
public static final char M_D_Y_FORMAT
```

This field indicates that the date format is month.day.year.

**MASTER:**

```
public static final String MASTER
```

This field indicates that the acting controller type is the master.

**MASTER\_AND\_FILE\_SERVER:**

```
public static final String MASTER_AND_FILE_SERVER
```

This field indicates that the acting controller type is the master and file server.

**MONOCHROME\_DISPLAY:**

```
public static final char MONOCHROME_DISPLAY
```

This field indicates that the display is a monochrome display.

**MONTH\_DAY\_YEAR\_FORMAT:**

```
public static final char MONTH_DAY_YEAR_FORMAT
```

This field indicates that the date format is month/day/year.

**PERIOD\_CONVENTION:**

```
public static final char PERIOD_CONVENTION
```

This field indicates that the monetary format uses the period convention.

**TOKENRING:**

```
public static final char TOKENRING
```

This field indicates that the LAN type is token ring.

***TWO\_SIGNIFICANT\_DIGITS:***

```
public static final char TWO_SIGNIFICANT_DIGITS
```

This field indicates that there are two significant digits after the decimal point.

***UNKNOWN\_DISPLAY:***

```
public static final char UNKNOWN_DISPLAY
```

This field indicates that the display type is unknown.

***ZERO\_SIGNIFICANT\_DIGITS:***

```
public static final char ZERO_SIGNIFICANT_DIGITS
```

This field indicates that there are no significant digits after the decimal point.

**ControllerStatusData Methods**

This section describes the methods for the ControllerStatusData class.

***isControllerOnActiveLan():*** This method tells you whether the controller is on an active LAN.

```
public boolean isControllerOnActiveLan()
```

**Returns**

**true** Controller is on an active LAN

**false** Controller is not on an active LAN

***getActingControllerType():*** This method informs you of the acting controller type.

```
public String getActingControllerType()
```

**Returns****CONTROLLER\_NOT\_ON\_LAN**

The acting controller is not on the LAN

**CONTROLLER\_ON\_LAN**

The acting controller is on the LAN

**ALTERNATE\_FILE\_SERVER**

The acting controller is the alternate file server

**FILE\_SERVER**

The acting controller is the file server

**ALTERNATE\_MASTER**

The acting controller is the alternate master

**MASTER**

The acting controller is the master

**MASTER\_AND\_FILE\_SERVER**

The acting controller is the master and file server

**ALT\_MASTER\_ALT\_FILESERVER**

The acting controller type is the alternate master and the alternate file server

***getAssignedConsole():*** This method tells you which console you are using.

```
public char getAssignedConsole()
```

**Returns**

**CONSOLE\_0**

Console is Console 0

**CONSOLE\_1**

Console is Console 1

**CONSOLE\_2**

Console is Console 2

**CONSOLE\_3**

Console is Console 3

**CONSOLE\_4**

Console is Console 4

**CONSOLE\_5**

Console is Console 5

**CONSOLE\_6**

Console is Console 6

**CONSOLE\_7**

Console is Console 7

**CONSOLE\_8**

Console is Console 8

***getControllerId():*** This method enables you to get the controller ID of your terminal.

```
public String getControllerId()
```

The method returns the store controller ID for this store controller.

***getDateFormat():*** This method enables you to get the format of the date.

```
public char getDateFormat()
```

**Returns****MONTH\_DAY\_YEAR\_FORMAT**

The format is returned as month/day/year

**DAY\_MONTH\_YEAR\_FORMAT**

The format is returned as day/month/year

**M\_D\_Y\_FORMAT**

The format is returned as month.day.year

**D\_M\_Y\_FORMAT**

The format is returned as day.month.year

***getDisplayType():*** This method enables you to get the type of display attached.

```
public char getDisplayType()
```

The method returns the display type as one of the following:

**UNKNOWN\_DISPLAY**

The display is unknown (for example, CMOS information is invalid)

**MONOCHROME\_DISPLAY**

The display is a monochrome display

**COLOR\_DISPLAY**

The display is a color display

***getLanConnectionType():*** This method enables you to identify the type of LAN with the store system.

```
public char getLanConnectionType()
```

The method returns one of the following:

**LAN\_NOT\_INSTALLED**

A LAN system is not installed

**TOKENRING**

The LAN type is token ring

**ETHERNET**

The LAN type is Ethernet

***getMasterControllerId():*** This method enables you to get the ID for the master controller for your terminal.

```
public String getMasterControllerId()
```

The method returns the ID of the master controller for your terminal.

***getMonetaryFormat():*** This method enables you to get the monetary format of your system.

```
public char getMonetaryFormat()
```

The method returns one of the following:

**PERIOD\_CONVENTION**

The monetary format uses the period convention (1,234,970.06)

**COMMA\_CONVENTION**

The monetary format uses the comma convention (1.234.970,06)

***getNumberOfDigitsAfterDecimal():*** This method allows you to get the number of significant digits configured.

```
public char getNumberOfDigitsAfterDecimal()
```

This method returns one of the following:

**ZERO\_SIGNIFICANT\_DIGITS**

There are no digits after the decimal point

**TWO\_SIGNIFICANT\_DIGITS**

There are two digits after the decimal point

***getNumberPrinterLinesPerPage():*** This method enables you to get the number of configured print lines per page.

```
public String getNumberPrinterLinesPerPage()
```

This method returns the number of printer lines per page. This number can range from 001 to 999.

***getNumberTerminalsConfigured():*** This method enables you to get the number of terminals configured.

```
public String getNumberTerminalsConfigured()
```

The method returns the number of configured terminals. The number can range from 001 through 999.

***getStoreNumber():*** This method enables you to get the store number.

```
public String getStoreNumber()
```

This method returns the store number.

**getTimeFormat():** This method enables you to get the time format of your system.

```
public char getTimeFormat()
```

The method returns one of the following:

**HOURL\_MINUTE\_SECOND\_FORMAT**

The time format is hour:minutes:seconds

**HMS\_TIME\_FORMAT**

The time format is hour.minutes.seconds

**getPrinterAssociatedWithConsole():** This method enables you to determine with which console the printer is associated.

```
public char getPrinterAssociatedWithConsole()
```

**Returns**

**CONSOLE\_1**

Console is Console 1

**CONSOLE\_2**

Console is Console 2

**CONSOLE\_3**

Console is Console 3

**CONSOLE\_4**

Console is Console 4

**CONSOLE\_5**

Console is Console 5

**CONSOLE\_6**

Console is Console 6

**CONSOLE\_7**

Console is Console 7

**CONSOLE\_8**

Console is Console 8

## Class multiapp.MultiAppStartup

The class `com.ibm.OS4690.multiapp.MultiAppStartup` is used to initiate multiple graphic Java applications in the terminal. If you want to run multiapp, the `MultiAppStartup` class should be configured as the default Java application for the terminal. See the *4690 OS: User's Guide* for additional information about using multiapp. Its usage is:

```
java com.ibm.OS4690.multiapp.MultiAppStartup <prop file>
```

## Startup Property File

The startup property file lists the applications that can be started from the Window Manager, as well as applications that should be started immediately. The `#` character at the beginning of a line indicates that the line is a comment. A single backslash (`\`) is interpreted as a line continuation character. If a backslash must be used for a property's value, then a double backslash (`\\`) must be used. For example, if an applications requires `c:\temp` as part of the argument then the `appArg` property would look like the following:

```
appN.appArgs=c:\\temp
```

The startup property file contains a series of blocks where **N** is the application name as follows:

**appN.appArgs**

= defines the arguments that are passed to the application (optional property)

**appN.className**

= defines the actual class whose main() is to be invoked (required property)

**appN.details**

= a brief message describing the applications to the user and is displayed in the Start Application dialog of the Window Manager (optional property)

**appN.displayName**

= defines the application name as it is displayed in the Window Manager dialog (required property)

**appN.GUIstartable**

= {true,false} determines if the application can be started from the Window Manager (optional property)

**appN.GUIstoppable**

= {true,false} determines if the application can be stopped from the Window Manager (optional property)

**appN.maxInstances**

= defines the maximum number of applications of this type that can be running (optional property)

**appN.onStartup**

= {true,false} determines if the application should automatically start up (if the application is the parent JVM, this parameter is ignored) (optional property)

**appN.parent**

= {true,false} defines this application as the parent (only 1 JVM can be the parent) (required property)

**appN.stdErr**

= is the name of the file that the standard error stream is redirected to (by default, the standard error stream is written to the console) (optional property)

**appN.stdOut**

= is the name of the file that the standard out stream is redirected to (by default, the standard out stream is written to the console) (optional property)

**appN.systemArgs**

= defines the arguments such as -classpath and -D properties that are passed to the system (optional property)

**Note:** systemArgs is not a valid property for the application that is defined to be the parent. System arguments for the parent must be specified in the invocation of MultiAppStartup. For example, if the parent application requires the -Dof flag, then the terminal's Java application definition would look like the following: -Dof com.ibm.OS4690.multiapp.MultiAppStartup r::c:/sample.prp

**appN.jvmID**

= {1, 2, 3, 4} specifies the unique ID for the JVM. This application is useful for performing Java method calls from a BASIC application.

**appN.defaultDisplay**

= {Primary, Secondary} specifies which video display that the JVM should display graphics on.

**Note:** This property is not legal for the parent application. It is not legal to specify this property for the primary application.

## Class multiapp.PropertyFileApplicationStartup

The class `multiapp.PropertyFileApplicationStartup` is an implementation of the `com.ibm.OS4690.multiapp.ApplicationStartup` interface. An instance of this class is created for each application defined in the multi-app properties file. An application can get access to these startups through `ApplicationManager.getApplicationStartups()`.

```
public class com.ibm.OS4690.multiapp.PropertyFileApplicationStartup implements
com.ibm.OS4690.multiapp.ApplicationStartup.

public class com.ibm.OS4690.multiapp.PropertyFileApplicationStartup
{
    //Methods
    public boolean isParent();
    public boolean onStartUp();
}
```

### PropertyFileApplicationStartup Methods

This section describes the methods for the `PropertyFileApplicationStartup` class.

**isParent():** Specifies whether the startup corresponds to the parent application.

```
public boolean isParent();
```

**onStartup():** Specifies whether the startup was executed when MultiApp was initialized.

```
public boolean onStartup();
```

## Class multiapp.ApplicationManager

Java applications can perform the same functions as the Window Manager dialog by using `multiapp.ApplicationManager`.

**Note:** Only the “parent JVM” can access this class.

```
public class com.ibm.OS4690.multiapp.ApplicationManager
{
    //Methods
    public static boolean isMultiAppEnabled();
    public static boolean isParent();
    public static ApplicationManager getApplicationManager();
    public Application getParentApplication();
    public Application getFocusedApplication();
    public Enumeration getRunningApplications();
    public void registerApplicationStartup( ApplicationStartup as );
    public void deregisterApplicationStartup( ApplicationStartup as );
    public Enumeration getApplicationStartups();
    public Application startApplication( ApplicationStartup as );
    public void stopApplication( Application app );
    public void switchTo( Application app );
    public void addApplicationListener( ApplicationListener listener );
    public void removeApplicationListener( ApplicationListener listener );
    public static getJvmID();
}
```

### ApplicationManager Methods

This section describes the methods for the `ApplicationManager` class.

**isMultiAppEnabled():** This method specifies whether the JVM is capable of supporting multiapp. This method can be called by either the parent JVM or the children JVMs.

```
public static boolean isMultiAppEnabled();
```

**isParent():** This method specifies whether the JVM is the parent. This method can be called by either the parent JVM or the children JVMs.



```
public static boolean isParent();
```

**ApplicationManager getApplicationManager():** This method returns an instance of an ApplicationManager. This method is available only to the parent JVM.

```
public static ApplicationManager getApplicationManager() throws  
UnauthorizedJVMException, MultiAppNotEnabledException
```

**Application getParentApplication():** This method returns the Application object for the parent JVM.

```
public Application getParentApplication();
```

**Application getFocusedApplication():** This method returns the Application object for the JVM with the currently focused Window.

```
public Application getFocusedApplication();
```

**Enumeration getRunningApplication():** This method returns an Enumeration of Application objects for all currently running JVMs.

```
public Enumeration getRunningApplication();
```

**registerApplicationStartup( ApplicationStartup as ):** This method registers an ApplicationStartup. If the ApplicationStartup specifies that it is GUIStartable, registering an ApplicationStartup with the ApplicationManager allows you to start the application described by the ApplicationStartup through the Window Manager.

```
public void registerApplicationStartup( ApplicationStartup as );
```

**deregisterApplicationStartup( ApplicationStartup as ):** This method deregisters the specified ApplicationStartup. After being deregistered, the Window Manager no longer provides the application as an option.

```
public void deregisterApplicationStartup( ApplicationStartup as );
```

**Enumeration getApplicationStartups():** This method provides an Enumeration of the ApplicationStartup objects that are registered with the ApplicationManager through the registerApplicationStartup().

```
public Enumeration getApplicationStartups();
```

**Application startApplication( ApplicationStartup as ):** This method starts the application that is specified by the ApplicationStartup. This method returns when the child JVM either fails to initialize or the child JVM successfully starts and initializes its graphics support.

```
public Application startApplication( ApplicationStartup as ) throws  
JVMStartupFailureException, TooManyInstancesException
```

**stopApplication( Application app ):** This method stops the application if the application is still running.

```
public void stopApplication( Application app ) throws ApplicationNotRunningException
```

**switchTo( Application app ):** This method raises all windows of the application to the front of the screen.

```
public void switchTo( Application app ) throws ApplicationNotRunningException
```

**addApplicationListener( ApplicationListener listener):** This method registers an ApplicationListener for getting ApplicationEvents.

```
public void addApplicationListener( ApplicationListener listener);
```

**removeApplicationListener( ApplicationListener listener ):** This method deregisters an ApplicationListener from getting ApplicationEvents.

```
public void removeApplicationListener( ApplicationListener listener );
```

**getJvmID():** Returns the integer ID for the JVM, which can be 1, 2, 3, or 4.

```
public static int getJvmID();
```

## Class multiapp.Application

This application corresponds to a JVM that is running under multiapp. This application is used to obtain the current status of an application

```
public class com.ibm.OS4690.multiapp.Application
{
    //Methods
    public ApplicationStartup getApplicationStartup();
    public boolean isParent();
    public String getDisplayName();
    public int getStatus();
    public int getExitValue();
    public String getDisplayName();
}
```

### Application Methods

This section describes the methods for the Application class.

**ApplicationStartup getApplicationStartup():** This method returns the ApplicationStartup that was used to start the JVM that this Application describes.

```
public ApplicationStartup getApplicationStartup();
```

**isParent():** This method returns true if this Application is describing the parent JVM.

```
public boolean isParent();
```

**String getDisplayName():** This method returns the name of the Application.

```
public String getDisplayName();
```

**getStatus():** This method returns one of two values; either the value Application.STATUS\_RUNNING or the value Application.STATUS\_STOPPED.

```
public getStatus();
```

**getExitValue():** This method returns JVM exited value. If the application is running, this method blocks until the application stops.

```
public getExitValue();
```

## Class multiapp.DisplayInfo

An application can get information concerning the dual display environment using this class.

```
public class com.ibm.OS4690.multiapp.DisplayInfo
{
    //Methods
    public static boolean isSecondaryPresent();
    public static int getDefaultDisplayType();
}
```

### DisplayInfo Methods

This section describes the static methods for the DisplayInfo class:

**isSecondaryPresent():** This method specifies whether or not a second video display is currently usable for graphics.

```
public static boolean isSecondaryPresent();
```

**getDefaultDisplayType():** This method specifies which video display is configured as the default for the JVM. It returns either DisplayInfo.DISPLAY\_TYPE\_PRIMARY or DisplayInfo.DISPLAY\_TYPE\_SECONDARY.

```
public static int getDefaultDisplayType();
```

## Interface multiapp.ApplicationStartup

ApplicationStartup describes how a JVM should be started. ApplicationStartup provides information such as the class to start and what arguments should be passed. See `ApplicationManager.startApplication(ApplicationStartup)` for additional information.

```
public interface com.ibm.OS4690.multiapp.ApplicationStartup
{
    //Methods
    public String getDisplayName();
    public String getDetails();
    public boolean isGUIStartable();
    public boolean isGUIStoppable();
    public int getMaxInstances();
    public int getInstances();
    public void incrementInstance();
    public String getSystemArgs();
    public String getClassname();
    public String getAppArgs();
    public String getStdOut();
    public String getStdErr();
    public int getJvmID()();
    public String getDefaultDisplay()();
}
```

### ApplicationStartup Methods

This section describes the methods for the ApplicationStartup interface.

***getDisplayName()***: This method displays the name of the application.

```
public String getDisplayName();
```

***getDetails()***: This method displays a brief message that describes the application.

```
public String getDetails();
```

***isGUIStartable()***: This method specifies whether the application can be started from the Window Manager.

```
public boolean isGUIStartable();
```

***isGUIStoppable()***: This method specifies whether the application can be stopped from the Window Manager.

```
public boolean isGUIStoppable();
```

***getMaxInstances()***: This method displays the maximum number of applications that are started from this ApplicationStartup and that can be running at one time.

```
public int getMaxInstances();
```

***getInstances()***: This method returns the current number of instances that are running.

```
public int getInstances();
```

***incrementInstance()***: This method is called when an application has been started that this object started.

```
public void incrementInstance() throws TooManyInstancesException;
```

***decrementInstance()***: This method is called when an application has been terminated that this object started.

```
public void decrementInstance();
```

***getSystemArgs()***: This method specifies system arguments that are to be passed to the JVM such as -classpath and -D properties.

```
public String getSystemArgs();
```

**getClassname():** This method displays the fully-qualified name of the class who's main() is to be invoked.

```
public String getClassname();
```

**getAppArgs():** This method specifies the application arguments (space-separated) that are to be passed to the application.

```
public String getAppArgs();
```

**getStdOut():** This method specifies the file to which stdout should be redirected. ApplicationStartup.NO\_REDIRECTION is used if no redirection is needed.

```
public String getStdOut();
```

**getStdErr():** This method specifies the file to which stderr should be redirected to. ApplicationStartup.NO\_REDIRECTION is used if no redirection is needed.

```
public String getStdErr();
```

**getJvmID():** This method returns the integer ID for the JVM to be created by this ApplicationStartup, which should be 1, 2, 3, or 4.

```
public int getJvmID();
```

**getDefaultDisplay():** This method specifies to which video display the created JVM should be targeted at. The value should be either "Primary" or "Secondary".

```
public String getDefaultDisplay();
```

## Interface multiapp.ApplicationListener

An ApplicationListener can register with the ApplicationManager through ApplicationManager.addApplicationListener() to receive ApplicationEvents.

```
public interface com.ibm.OS4690.multiapp.ApplicationListener
{
    //Methods
    public void applicationEvent(ApplicationEvent ae);
}
```

## ApplicationStartupListener Method

This section describes the method for the ApplicationListener interface.

**applicationEvent:** This method describes a status change for an application. Notification is given for the following events: application starting, application stopping, application gaining keyboard focus, application losing keyboard focus.

```
public class comm.ibm.OS4690.multiapp.ApplicationEvent;
```

## Class multiapp.ApplicationEvent

ApplicationEvent describes a status change for an application. Notification is given for the following events: application starting, application stopping, application gaining keyboard focus, application losing keyboard focus.

```
public class com.ibm.OS4690.multiapp.ApplicationEvent
{
    //Methods
    public Application getApplication();
    public int getEventType();
}
```

## ApplicationEvent Methods

This section describes the static methods for the ApplicationEvent class.

**getApplication():** This static method returns the Application object in which this event is associated.

```
public Application getApplication();
```

**getEventType():** This static method returns one of the following constants: APP\_STARTED, APP\_STOPPED, APP\_GAINED\_FOCUS, APP\_LOST\_FOCUS.

```
public int getEventType();
```

## Class TerminalApplicationServices

Java applications use a class called TerminalApplicationServices to request ADXSERVE functions. The methods in this section must be run in a terminal.

```
public class com.ibm.OS4690.TerminalApplicationServices
    extends Object
    implements Def4690
{
    //Methods
    public static void disableIPL();
    public static void disableStorageRetention();
    public static void dumpSystemStorage();
    public static TerminalStatusData getTerminalStatus();
    public static boolean isTerminal();
    public static void logError(char, int, int, int, byte[]);
    public static void powerOffLocalMachine(String);
    public static void setIPL();
    public static void setStorageRetention();
    public static void setTermJavaKeysToAwt();
    public static void setTermJavaKeysToJavaPOS();
    public static void switchToJavaScreen();
    public static void switchToTerminalScreen();
    public static void switchToSystemFunctionScreen();
}
```

## TerminalApplicationServices Methods

This section describes the methods for the TerminalApplicationServices class.

**disableIPL():** This method prevents the automatic reload that might occur when a terminal comes online.

```
public static void disableIPL() throws FlexosException;
```

**disableStorageRetention():** This method disables storage retention in all of the terminals on the TCC Network of the store controller requesting the disable.

```
public static void disableStorageRetention() throws FlexosException;
```

**dumpSystemStorage():** The dumpSystemStorage method causes all of the storage in the terminal to be dumped to a disk file named ADX\_SDT1:ADXCSLTF.DAT in the root directory.

```
public static void dumpSystemStorage() throws FlexosException;
```

**getTerminalStatus():** This method returns an object of type TerminalStatusData that has various methods to gather status information about the terminal. Access to the status information is obtained by invoking the TerminalStatusData object's methods. See "Class TerminalStatusData" on page 530 for information on the TerminalStatusData object's methods.

```
public static TerminalStatusData getTerminalStatus() throws FlexosException;
```

This method throws a FlexosException exception with a return code of 0x80004085 if you try to call getTerminalStatus from a store controller application.

**isTerminal():** This method performs a query to determine if the application is running on a terminal.

```
public static boolean isTerminal();
```

**logError(char, int, int, int, byte[]):** This method is used to log an error in the application error log. The message should be in file ADXCSOZF.DAT. The application name is automatically included in the log entry.

```
public static void logError(char msggrp,
                           int msgnum,
                           int severity,
                           int event,
                           byte[] unique) throws FlexosException
                                           InvalidParameterException
```

#### Parameters:

##### msggrp

Message group character, which is unique for each product and should be in the range of J to S.

##### msgnum

Message number, if any. If the message number is zero, no message is displayed. This number is converted to three printable decimal digits.

##### severity

Number ranging from 1 to 5 that indicates the importance of the message. The most important is severity 1.

**event** Value assigned by the application that is used to indicate why the error is being logged. This number is converted to three printable decimal digits.

##### unique

Byte array to be included in the message. The maximum length is 10 bytes.

**powerOffLocalMachine(String):** This method is used to power off a local machine. The machine can be a controller or terminal, provided the controller or terminal is in the 469x or SurePOS family. This method should be invoked on the machine that is to be powered off. The controller or terminal is powered-off immediately when this method is called. The time in the string provided by the caller as input is the power-on time.

```
public static void powerOffLocalMachine(String powerData) throws FlexosException;
```

#### Parameters:

##### powerData

Power on time (DDHHMM), where:

**DD** Day of the month (01–31)

**HH** Hours (00–24)

**MM** Minutes (00–59)

The following FlexosException exception return codes are thrown by the method:

**-1016** The machine to be powered off is not in the 4690 family or the machine is a controller/terminal environment.

**-1017** A date or time that is not valid was specified.

**-1018** The controller ID is not valid or is not the active controller.

**-1020** BIOS driver open failed.

**-1023** The programmable power request is pending. Either programmable power has been disabled or an application has set the NO-IPL flag.

**setIPL():** This method is used to enable the terminal to IPL. If the terminals were IPLed earlier, but could not because the user had disabled the IPL, calling this method causes the terminal to IPL. This function is

also used to enable programmable power. It allows the terminal to accept power-off commands. If a previous power-off command has been issued while programmable power has been disabled, and the power-on time has not passed, this function causes the terminal to power off.

```
public static void setIPL() throws FlexosException;
```

**setStorageRetention():** This method enables storage retention in all of the terminals on the TCC Network of the store controller requesting the enable.

```
public static void setStorageRetention() throws FlexosException;
```

## Switching the terminal Java keystroke destination for shared ANPOS keyboards

There are two ways that a terminal Java application can receive input from an ANPOS keyboard, by instantiating an instance of the JavaPOS class POSKeyboard, or by using the AWT Event Queue. If JavaPOS is selected as the Java keystroke destination, all keys on the ANPOS keyboard are delivered as DataEvents to the POSKeyboard. If Java AWT is selected, keystrokes are delivered to the AWT Event Queue, but all POS-specific keys are ignored.

For Java 2 terminal applications, the default Java keystroke destination is defined in 4690 Generic Terminal Configuration **Device Characteristics->Keyboards->Select** your Java application keyboard source. For Java 6 terminal applications (Enhanced Mode systems only), the default Java keystroke destination is always Java AWT.

The following two methods enable you to switch the Java keystroke destination:

```
setTermJavaKeysToAwt()  
setTermJavaKeysToJavaPOS()
```

For Java 2 terminal applications, the terminal Java screen must be displayed for the Java keystroke destination to be in effect.

For Java 6 terminal applications, setting the Java keystroke destination affects the destination of keystrokes no matter which screen is displayed. If Java AWT is selected, keystrokes are delivered to the AWT Event Queue for the Java 6 POS application or delivered to other 4690 applications depending on which application has focus.

If JavaPOS is selected, keystrokes are sent only to 4690 and are delivered as DataEvents to the POSKeyboard. To ensure keystrokes are sent to the correct 4690 application, a Java 6 POS application requiring JavaPOS must select JavaPOS as the Java keystroke destination only when it has focus. The application is responsible for switching the Java keystroke destination when it gains and loses focus. When it gains focus, it must set the Java keystroke destination to JavaPOS by calling setTermJavaKeysToJavaPOS(). When it loses focus, it must set the Java keystroke destination to Java AWT by calling setTermJavaKeysToJavaAWT(). The keyboard mode is automatically set to AWT when a Java6 application ends.

### setTermJavaKeysToAwt():

This method enables you to set Java AWT as the terminal Java keystroke destination. If Java AWT is selected, then System Functions (S1, key code, S2) are not available from the terminal Java application screen.

```
public static void setTermJavaKeysToAWT()throws FlexosException;
```

The following example sets Java AWT as the Java keystroke destination and displays the terminal Java screen:

```
TerminalApplicationServices.setTermJavaKeysToAwt();  
TerminalApplicationServices.switchToJavaScreen();
```



### setTermJavaKeysToJavaPOS():

This method enables you to set JavaPOS as the terminal Java keystroke destination.

```
public static void setTermJavaKeysToJavaPOS()throws FlexosException;
```

The following example sets JavaPOS as the Java keystroke destination and displays the terminal Java screen:

```
TerminalApplicationServices.setTermJavaKeysToJavaPOS();  
TerminalApplicationServices.switchToJavaScreen();
```

## Switching the terminal Java, terminal application, and enhanced mode graphical extensions screens

On a controller or terminal, the system must be on the terminal side for these APIs to display either the terminal Java screen (Java 2 or Java 6), the terminal application screen, or the enhanced mode graphical extensions screen.

### switchToJavaScreen():

This method is used to switch the terminal video so that the terminal Java screen is displayed. It has the same effect as typing **Alt+SysRq+J** on an ANPOS or standard keyboard, or typing S1,5,S2 on the POS keyboard when the terminal application screen is displayed.

```
public static void switchToJavaScreen()throws FlexosException;
```

### switchToTerminalScreen():

This method is used to switch the terminal video so that the terminal application screen is displayed. It has the same effect as typing **Alt+SysRq+T** on an ANPOS or standard keyboard, or typing S1,5,S2 on the POS keyboard when the terminal Java screen is displayed.

```
public static void switchToTerminalScreen()throws FlexosException;
```

### switchToXWindowScreen():

This method is used to switch the terminal video so that the enhanced mode graphical extensions screen is displayed. It has the same effect as typing **Alt+SysRq+X** on an ANPOS or standard keyboard, or typing S1,5,S2 on the POS keyboard. This method is not supported on Classic systems.

```
public static void switchToXWindowScreen() throws FlexosException;
```

## Class TerminalStatusData

The getTerminalStatus method returns an object of type *TerminalStatusData* that has various methods for gathering status information about the terminal. Access to the status information is obtained by invoking the TerminalStatusData object's methods.

```
public class com.ibm.OS4690.TerminalStatusData  
    extends Object  
{  
    //Fields  
    public static final char ANDISPLAY  
    public static final char ANDISPLAY2  
    public static final char ANDISPLAY3  
    public static final char COMMA_CONVENTION  
    public static final char CONTRÖLLER_TERMINAL  
    public static final char D_M_Y_FORMAT  
    public static final char DAY_MONTH_YEAR_FORMAT  
    public static final char HMS_TIME_FORMAT  
    public static final char HOUR_MINUTE_SECONDS_FORMAT  
    public static final char M_D_Y_FORMAT  
    public static final char MONTH_DAY_YEAR_FORMAT  
    public static final String NVRAM_SIZE_1K  
    public static final String NVRAM_SIZE_16K  
    public static final char PERIOD_CONVENTION  
    public static final char TERMINAL  
    public static final char TERMINAL_4683
```



```

public static final char TERMINAL_4684
public static final char TERMINAL_4693
public static final char TERMINAL_4694
public static final char TWO_SIGNIFICANT_DIGITS
public static final char VDISPLAY
public static final char VDISPLAY2
public static final char VGA_VIDEO_DISPLAY
public static final char VIDEO_DISPLAY_FOR_4683
public static final char ZERO_SIGNIFICANT_DIGITS
public static final char KEYBOARD_UNINIT      = ' ';
public static final char KEYBOARD_50K          = ' ';
public static final char KEYBOARD_133K         = ' ';
public static final char KEYBOARD_ANPOS        = ' ';
public static final char KEYBOARD_KEYPAD       = ' ';
public static final char KEYBOARD_DBCS_V       = ' ';
public static final char KEYBOARD_DBCS_VI      = ' ';
public static final char KEYBOARD_UNKNOWN      = ' ';

//Methods
public char getApplicationEnvironment()
public char getDateFormat()
public String getDefaultApplicationName()
public byte getEcLevelof4610MicroCode()
public char getMonetaryFormat()
public char getNumberOfDigitsAfterDecimal()
public String getNVRAMSize()
public String getPartnerTerminalAddress()
public String getStoreControllerId()
public String getStoreNumber()
public char getSystemDisplayType()
public String getTerminalNumber()
public char getTerminalType()
public char getTimeFormat()
public char getVideoAdapterType()
public boolean isFlipperPresentOn4610()
public boolean isLoopInBackup()
public boolean isMicrPresentOn4610()
public boolean isPrinterA4610()
public boolean isPrinterATi1OrTi2()
public boolean isSBCSInUseOn4610()
public boolean isStorageRetentionEnabled()
public boolean isTerminalOnLine()
public boolean isTerminalPoweredOn()
public boolean isTerminalTheMaster()
}

```

## TerminalStatusData Fields

This section describes the fields for the TerminalStatusData class.

### **ANDISPLAY:**

```
public static final char ANDISPLAY
```

This field indicates that the current video display adapter is ANDISPLAY.

### **ANDISPLAY2:**

```
public static final char ANDISPLAY2
```

This field indicates that the current video display adapter is ANDISPLAY2.

### **ANDISPLAY3:**

```
public static final char ANDISPLAY3
```

This field indicates that the current video display adapter is ANDISPLAY3.

**COMMA\_CONVENTION:**

```
public static final char COMMA_CONVENTION
```

This field indicates that the monetary format uses a comma.

**CONTROLLER\_TERMINAL:**

```
public static final char CONTROLLER_TERMINAL
```

This field indicates that the application is running in a controller/terminal.

**D\_M\_Y\_FORMAT:**

```
public static final D_M_Y_FORMAT
```

This field indicates that the date format is day.month.year.

**DAY\_MONTH\_YEAR\_FORMAT:**

```
public static final char DAY_MONTH_YEAR_FORMAT
```

This field indicates that the date format is day/month/year.

**HMS\_TIME\_FORMAT:**

```
public static final char HMS_TIME_FORMAT
```

This field indicates that the time format is hour.minute.seconds.

**HOURL\_MINUTE\_SECONDS\_FORMAT:**

```
public static final char HOUR_MINUTE_SECONDS_FORMAT
```

This field indicates that the time format is hour:minute:seconds.

**M\_D\_Y\_FORMAT:**

```
public static final char M_D_Y_FORMAT
```

This field indicates that the date format is month.day.year.

**MONTH\_DAY\_YEAR\_FORMAT:**

```
public static final char MONTH_DAY_YEAR_FORMAT
```

This field indicates that the date format is month/day/year.

**NVRAM\_SIZE\_1K:**

```
public static final String NVRAM_SIZE_1K
```

This field indicates that the size for the Hard Totals (NVRAM) is 1 KB.

**NVRAM\_SIZE\_16K:**

```
public static final String NVRAM_SIZE_16K
```

This field indicates that the size for the Hard Totals (NVRAM) is 16 KB.

**PERIOD\_CONVENTION:**

```
public static final char PERIOD_CONVENTION
```

This field indicates that the monetary format uses a period.

**TERMINAL:**

```
public static final char TERMINAL
```

This field indicates that application is running in a terminal.

**TERMINAL\_4683:**

```
public static final char TERMINAL_4683
```

This field indicates that the terminal currently configured is a 4683 machine.

**TERMINAL\_4684:**

```
public static final char TERMINAL_4684
```

This field indicates that the terminal currently configured is a 4684 machine.

**TERMINAL\_4693:**

```
public static final char TERMINAL_4693
```

This field indicates that the terminal currently configured is a 4693 machine.

**TERMINAL\_4694:**

```
public static final char TERMINAL_4694
```

This field indicates that the terminal currently configured is a 4694 machine.

**TERMINAL\_4800:**

```
public static final char TERMINAL_4800
```

This field indicates that the terminal currently configured is a SurePOS 300/700 Series machine.

**TWO\_SIGNIFICANT\_DIGITS:**

```
public static final char TWO_SIGNIFICANT_DIGITS
```

This field indicates that there are two significant digits after the decimal.

**VDISPLAY:**

```
public static final char VDISPLAY
```

This field indicates that the current video display adapter is VDISPLAY.

**VDISPLAY2:**

```
public static final char VDISPLAY2
```

This field indicates that the current video display adapter is VDISPLAY2.

**VGA\_VIDEO\_DISPLAY:**

```
public static final char VGA_VIDEO_DISPLAY
```

This field indicates that the video display adapter is a VGA display.

**VIDEO\_DISPLAY\_FOR\_4683:**

```
public static final char VIDEO_DISPLAY_FOR_4683
```

This field indicates that the video display adapter is a 4683 Video Display Feature A.

### ***ZERO\_SIGNIFICANT\_DIGITS:***

```
public static final char ZERO_SIGNIFICANT_DIGITS
```

This field indicates that there are no significant digits after the decimal point.

## **TerminalStatusData Methods**

This section describes the methods for the TerminalStatusData class.

***getApplicationEnvironment():*** This method indicates whether the application is running in a terminal or controller/terminal.

```
public char getApplicationEnvironment()
```

The method returns the one of the following:

### **TERMINAL**

The application is running in a terminal

### **CONTROLLER\_TERMINAL**

The application is running in a controller/terminal

***getDateFormat():*** This method enables you to get the date format.

```
public char getDateFormat()
```

The method returns the following information:

### **MONTH\_DAY\_YEAR\_FORMAT**

The format is returned as month/day/year

### **DAY\_MONTH\_YEAR\_FORMAT**

The format is returned as day/month/year

### **M\_D\_Y\_FORMAT**

The format is returned as month.day.year

### **D\_M\_Y\_FORMAT**

The format is returned as day.month.year

***getDefaultApplicationName():*** This method returns the default application name from terminal configuration.

```
public String getDefaultApplicationName()
```

***getEcLevelof4610MicroCode():*** This method tells you the EC level of the 4610 printer's micro code. If this method is called without a 4610 printer being attached a FlexosException exception is thrown.

```
public byte getEcLevelof4610MicroCode() throws FlexosException
```

The method returns the EC level of the 4610 printer's micro code. If a 4610 printer is not present and this method is called, a FlexosException exception with a return code of 0x00 is thrown.

***getMonetaryFormat():*** This method enables you to get the monetary format of your system.

```
public char getMonetaryFormat()
```

The method returns one of the following:

### **PERIOD\_CONVENTION**

The monetary format uses the period convention (1,234,970.06)

### **COMMA\_CONVENTION**

The monetary format uses the comma convention (1.234.970,06)

***getNumberOfDigitsAfterDecimal():*** This method allows you to get the number of significant digits configured.

```
public char getNumberOfDigitsAfterDecimal()
```

This method returns one of the following:

**ZERO\_SIGNIFICANT\_DIGITS**

There are no digits after the decimal point

**TWO\_SIGNIFICANT\_DIGITS**

There are two digits after the decimal point

***getNVRAMSize():*** This method tells you the size for the Hard Totals (NVRAM). The possible return values are either 1K or 16K.

```
public String getNVRAMSize()
```

The method returns one of the following:

**NVRAMSize1K**

The Hard Totals size is 1 KB (4683)

**NVRAMSize16K**

The Hard Totals size is 16 KB (4693/4694)

***getPartnerTerminalAddress():*** This method returns the address of the Mod2 terminals.

```
public String getPartnerTerminalAddress()
```

***getStoreControllerId():*** This method returns the ID of the store controller for your terminal.

```
public String getStoreControllerId()
```

***getStoreNumber():*** This method returns the store number.

```
public String getStoreNumber()
```

***getSystemDisplayType():*** This method tells you the current video display adapter present.

```
public char getSystemDisplayType()
```

The method returns one of the following:

**ANDISPLAY**

Display named ANDISPLAY is the system display

**ANDISPLAY2**

Display named ANDISPLAY2 is the system display

**VDISPLAY**

Display named VDISPLAY is the system display

**VDISPLAY2**

Display named VDISPLAY2 is the system display

**ANDISPLAY3**

System named ANDISPLAY3 is the system display

***getTerminalNumber():*** This method returns the terminal number.

```
public String getTerminalNumber()
```

***getTerminalType():*** This method tells you the type of terminal currently configured. These are the possible terminal types:

- 4683

- 4684
- 4693
- 4694
- SurePOS 300/700 Series

```
public char getTerminalType()
```

This method returns one of the following values:

**TERMINAL\_4693**

The terminal is a 4693 terminal

**TERMINAL\_4694**

The terminal is a 4694 terminal

**TERMINAL\_4683**

The terminal is a 4683 terminal

**TERMINAL\_4684**

The terminal is a 4684 terminal

**TERMINAL\_4800**

The terminal is a SurePOS 300/700 Series terminal

***getTimeFormat():*** This method enables you to get the time format of your system.

```
public char getTimeFormat()
```

The method returns one of the following values:

**H\_M\_S\_TIME\_FORMAT**

The time format is hour:minutes:seconds

**HMS\_TIME\_FORMAT**

The time format is hour.minutes.seconds

***getVideoAdapterType():*** This method enables you to determine whether the video display adapter is a 4683 video display feature A or a VGA display.

```
public char getVideoAdapterType()
```

The method returns one of the following values:

**VIDEO\_DISPLAY\_FOR\_4683**

The video display adapter is a 4683 video display feature A

**VGA\_VIDEO\_DISPLAY**

The video display is a VGA display

***isFlipperPresentOn4610():*** This method determines whether the attached 4610 printer has a check flipper.

```
public boolean isFlipperPresentOn4610()
```

The method returns one of the following values:

**true** Check flipper is present

**false** Check flipper is not present

***isLoopInBackup():***

**Note:** Store Loop support was removed in 4690 OS V6R3

This method determines whether the loop is in backup or not.

```
public boolean isLoopInBackup()
```

The method returns one of the following values:

**true**     Loop is in backup

**false**    Loop is not in backup

***isMicrPresentOn4610():*** This method determines whether the attached 4610 printer has a MICR reader.

```
public boolean isMicrPresentOn4610()
```

The method returns one of the following values:

**true**     MICR is present

**false**    MICR is not present

***isPrinterA4610():*** This method tells you if a 4610 printer is attached.

```
public boolean isPrinterA4610()
```

The method returns one of the following values:

**true**     Printer is a 4610

**false**    Printer is not a 4610

***isPrinterATi1OrTi2():*** This method determines if the attached 4610 printer is in the TI1 or TI2 family.

```
public boolean isPrinterATi1OrTi2()
```

The method returns one of the following values:

**true**     Printer is in the TI1 family

**false**    Printer is in the TI2 family

***isSBCSInUseOn4610():*** This method tells you whether the attached 4610 printer uses the single-byte character set (SBCS).

```
public boolean isSBCSInUseOn4610()
```

The method returns one of the following values:

**true**     Single-Byte Character Set is used

**false**    Single-Byte Character Set is not used

***isStorageRetentionEnabled():*** This method enables you to determine if storage retention is enabled.

```
public boolean isStorageRetentionEnabled()
```

The method returns one of the following values:

**true**     Storage Retention is enabled

**false**    Storage Retention is disabled

***isTerminalOnLine():*** This method enables you to determine if the terminal is online.

```
public boolean isTerminalOnline()
```

The method returns one of the following values:

**true** Terminal is online

**false** Terminal is offline

***isTerminalPoweredOn():*** This method enables you to determine whether the terminal is powered on.

```
public boolean isTerminalPoweredOn()
```

The method returns one of the following values:

**true** Terminal is powered off (Always on for Mod1; can be on or off for Mod2)

**false** Terminal is powered on

***isTerminalTheMaster():*** This method enables you to determine if the terminal is the master (Mod1).

```
public boolean isTerminalTheMaster()
```

The method returns one of the following values:

**true** Terminal is master (Mod1)

**false** Terminal is not the master

***getKeyboardType():*** The method returns the POS keyboard type.

```
public char getKeyboardType()
```

The method returns one of the following values:

**KEYBOARD\_UNINIT**

uninitialized

**KEYBOARD\_50K**

50 key keyboard

**KEYBOARD\_MOD\_67K**

Modular 67 key keyboard

**KEYBOARD\_MOD\_67K\_LCD**

Modular 67 key LCD keyboard

**KEYBOARD\_133K**

133 key matrix keyboard

**KEYBOARD\_ANPOS**

ANPOS keyboard

**KEYBOARD\_MOD\_ANPOS**

Modular ANPOS keyboard

**KEYBOARD\_KEYPAD**

4820 keypad

**KEYBOARD\_DBCS\_V**

DBCS keyboard V

**KEYBOARD\_DBCS\_VI**

DBCS keyboard VI

**KEYBOARD\_UNKNOWN**

unknown/other keyboard



## Class POSPlatform

Virtual terminal sessions can run on controllers as well as terminals. Some Java applications could be running either on a real 4690 terminal, or in a virtual terminal session on a 4690 controller. These applications may use the `POSPlatform.convertFilename()` method for controller-based files, so that they need not know in which of these environments they are running. The class resides in `OS4690.zip`. (The user should use only the documented methods in this class. Any other public methods are subject to change.)

```
public class com.ibm.retail.oscommon.POSPlatform
{
    //Methods
    public static String convertFilename(String)
}
```

### POSPlatform Methods

This section describes the methods for the `POSPlatform` class.

`convertFilename(String)`: When virtual terminals run on a controller and try to open a file whose name starts with "R::", the open will throw a `FileNotFoundException`. To avoid this problem, Java applications that could be running either on a real 4690 terminal, or in a virtual terminal session on a 4690 controller, should convert controller-based file names by using this method. This method returns a `String` object representing the correct file name for the point of sale platform on which the application is running.

```
public static String convertFilename(String fileName)
```

#### Parameters:

##### **fileName**

The file name to be converted.

---

## Using I/O Redirection to Java

The 4690 Operating System provides support to enable you to redirect an existing application's I/O from `ANDisplay1`, `ANDisplay2`, the `IOProcessor` and the printer Cash Receipt Station, printer Document Insert Station, printer Summary Journal Station, and Magnetic Stripe Reader devices to a Java Application. This support helps you develop a Java GUI that seamlessly communicates with your legacy application. There are several pieces that work together to allow you to communicate from either C or CBASIC to Java

This implementation was designed to hide the complexity of the Operating System, while providing a straight forward Java API to all types of essential calls for each terminal device. The section that follows contains the name of each component, a description of its role in the system, and the interface you must use to communicate with it.

## Handlers and Monitors

There are two types of implementations possible from the provided interfaces. One is a handler, the other is a monitor.

### Handlers

The operating system provides handler interfaces for these devices:

- `ANDisplay1`
- `ANDisplay2`
- `IOProcessor`
- `MSR`
- `POSPrinter`

When you choose to implement the handler interface for any of these devices, you assume all responsibility for that device and must handle all the communication to and from the application as that hardware would have done.

## Monitors

A monitor is provided as an interface for the POSPrinter Cash Receipt Station. With a CashReceiptMonitor configured, you can receive a copy of the data being sent to the CashReceiptStation device. The physical printer is still connected and is responsible for handling all communications back to the application.

**CashReceiptStation:** Previously, support was limited to monitoring the data only. However, now a Java application can function either as a handler or as a monitor for the Cash Receipt Station. When the Java application is configured as a handler, it is responsible for processing the data and returning the correct return code to the C or CBASIC application as in all other handlers.

**Note:** The Cash Receipt Station can be redirected to either a monitor or a handler, but **cannot** be redirected to both. This distinction is very important.

---

## Configuring Redirected Devices

One of the first steps is to configure which devices are supported in Java, and which you are going to redirect to the Java Application. This is done through Terminal Configuration in the Terminal Device Group panels. You are prompted with this question:

```
Will a terminal using this device group run a Java application and will
that application need to receive redirection input from I/O devices?_
```

After selecting **1** for **yes** to the question, this panel appears:

Please place an X next to the devices you wish to redirect to Java.

```
___ ANDisplay1
___ ANDisplay2
___ IOProcessor
___ CashReceiptStation
___   Monitor
___   Handler
___ Document Insert Station
___ Summary Journal Station
___ MSR Handler
```

See the *4690 OS Planning, Installation, and Configuration Guide* for detailed instructions on configuring devices for Java and redirection.

---

## Class DeviceManager

Although the old static API for interfacing with the DeviceManager continues to be supported, it has been deprecated. A reference to a DeviceManagerInterface should be obtained either through DeviceManager.singleton() for a true terminal environment, or through DeviceManager.createDeviceManager() for multiple emulated terminals in a controller environment. DeviceManagerInterface can be used to perform all functions supported by the old static API.

DeviceManager is a Java class that resides in OS4690.ZIP and is the main communication mechanism between C or BASIC and Java. The DeviceManager class is an extension of the Object class. It is a static class that manages access to input and output that has been redirected from an existing application. It also provides the user with registration capability for each type of handler that is configured.

Registration simply means passing the DeviceManager class an object that has implemented the interface for the type of device from which you are trying to receive I/O.

After you have registered all redirected devices by passing a reference to the handler, you must call a method to tell the DeviceManager that you are ready to begin communicating with the driving application. The method called to do this is `setRegistrationComplete()`.

After this method is called, the DeviceManager begins communicating with the driving C or BASIC application, and uses the references you have passed in for each handler to complete calls to the application.

After a device is registered a terminal reload is required to change its registration.

The following methods are provided in the DeviceManager interface and allow you to redirect I/O for one or all devices:

- `setANDisplay1Handler(ANDisplayHandler display1)`
- `setANDisplay2handler(ANDisplayHandler display2)`
- `setCashReceiptMonitor(CashReceiptMonitor cr)`
- `setIOPHandler(IOPHandler IOP)`
- `setExtendedIOPHandler(ExtendedIOPHandler iop)`
- `setPOSPrinterHandler(POSPrinterHandler prn)`
- `setMSRHandler(MSRHandler msr)`
- `setDeviceRegistrationComplete()`

## DeviceManager Methods

### **singleton()**

This method returns a reference to a `DeviceManagerInterfaceMultiJVM`, which is an extension of the `DeviceManagerInterface` appropriate for a true terminal environment. This method is preferred for interfacing with the DeviceManager.

```
public static DeviceManagerInterfaceMultiJVM singleton();
```

### **createDeviceManager(int terminalID)**

This method returns a reference to a `DeviceManagerInterfaceTSS`, which is an extension of the `DeviceManagerInterface` appropriate for an environment with multiple emulated terminals.

```
public static DeviceManagerInterfaceTSS createDeviceManager(int terminalID);
```

### **IOPDataAvailable()**

This method notifies the DeviceManager that IOP Data is available.

```
public static void IOPDataAvailable()
```

### **MSRDataAvailable()**

This method notifies the DeviceManager that MSR data is available.

```
public static void MSRDataAvailable()
```

### **POSPrinterDataAvailable(int nStation)**

This method notifies the DeviceManager that data is available from the specified printer station.

```
public static void POSPrinterDataAvailable(int nStation)
```

Parameter: *nStation* can be one of the following:

- `DeviceManagerConst.PS_CASH_RECEIPT`
- `DeviceManagerConst.PS_DOCUMENT_INSERT`
- `DeviceManagerConst.PS_SUMMARY_JOURNAL`

### **run()**

```
public void run()
```

### **setANDisplay1Handler(ANDisplayHandler)**

This method registers the ANDisplay1Handler with the DeviceManager.

```
public static int setANDisplay1Handler(ANDisplayHandler display)
```

### **setANDisplay2Handler(ANDisplayHandler)**

This method registers the ANDisplay2Handler with the DeviceManager.

```
public static int setANDisplay2Handler(ANDisplayHandler display)
```

### **setCashReceiptMonitor (CashReceiptMonitor)**

This method registers the CashReceiptMonitor with the DeviceManager. Use this method when the Cash Receipt station has been configured as a monitor

```
public static int setCashReceiptMonitor(CashReceiptMonitor cr)
```

### **setIOPHandler(IOPHandler)**

This method registers the IOPHandler with the DeviceManager.

```
public static void setIOPHandler(IOPHandler iop)
```

### **setExtendedIOPHandler (ExtendedIOPHandler)**

This method registers the ExtendedIOPHandler with the DeviceManager.

```
public static int setExtendedIOPHandler(ExtendedIOPHandler iop)
```

### **setPOSPrinterHandler (POSPrinterHandler)**

This method registers the POSPrinterHandler with the DeviceManager. The POSPrinterHandler can handle all three printer stations: Cash Receipt, Document Insert, and Summary Journal.

```
public static int setPOSPrinterHandler(POSPrinterHandler iop)
```

### **setMSRHandler(MSRHandler)**

This method registers the MSRHandler with the DeviceManager.

```
public static int setMSRHandler(MSRHandler iop)
```

### **setDeviceRegistrationComplete()**

This method is called when all device registration is complete and when the DeviceManager has a reference for all redirected devices. After setDeviceRegistrationComplete has been called, the operating system ignores further calls to any of the registration methods, and discards any reference passed in.

```
public static int setDeviceRegistrationComplete()
```

## **DeviceManager Tracing**

DeviceManager tracing can be configured using a property file. The property file must exist on the classpath in a directory structure like com/ibm/retail/DevMgr/logging. The name of the property file must be logging.properties. By default, OS4690.zip will contain com/ibm/retail/DevMgr/logging/logging.properties.

The properties file contains these properties:

#### **terminals**

This property defines which terminals should have tracing enabled. Valid values for this property are *all* (for all terminals), *none* (for no terminals), or a range of terminals. An example of a range of terminals is 1-5, 30-40, 45. The default value of this property is *none*.

#### **AllDevices**

This property allows tracing to be enabled or disabled for all devices. Valid values are *true* and *false*.

Tracing can be enabled and disabled on a per-device basis. Properties exist for each device. Each property can be set to *true* or *false*. These are the properties that can be set:

- IOProcessor

- ANDisplay1
- ANDisplay2
- ANDisplay3
- CashReceipt
- MethodInvocation
- MSR
- CashDrawer
- DocumentInsert
- SummaryJournal
- ToneIndicator
- NVRAM
- SerialPort1
- SerialPort2
- SerialPort3
- SerialPort4
- Scale
- Video
- ApplicationServices

## Overriding a Property

Any property in the properties file can be overridden by a system property. To override a setting in the properties file, define a system property with the desired property name prepended with "com.ibm.retail.DevMgr.logging". For example, to turn on tracing for the scale without modifying a properties file, you could specify `-Dcom.ibm.retail.DevMgr.logging.Scale=true` in the JVM invocation.

## Trace Output

If the JVM running the DeviceManager is a version prior to 1.4, then the trace will go to a file named `R::C:/DM###.LOG`, where `###` is the terminal number.

If the JVM running the DeviceManager is 1.4 or higher, then the JDK logging capabilities are used. The name of the logger is "com.ibm.retail.DevMgr.###", where `###` is the terminal number. By default the logger will write output to stdout. (For details on how to change this behavior, refer to documentation on JDK `java.util.logging`.)

The handler configured for DeviceManager logging must be set to level FINE for the device tracing to be seen. By default, the handler is `java.util.logging.ConsoleHandler`. This is defined by the property "handlers" in `m:/java2/jre/lib/logging.properties`. The default level for the `ConsoleHandler` is INFO, which means the DeviceManager trace will not be active. In order to activate the DeviceManager trace, you must set the property `java.util.logging.ConsoleHandler.level` in `m:/java2/jre/lib/logging.properties` to FINE.

---

## Interface DeviceManagerInterface

`DeviceManagerInterface` is an interface for interacting with an instance of a `DeviceManager`. An instance of a `DeviceManagerInterface` can be obtained through `DeviceManager.singleton()` or `DeviceManager.createDeviceManager()`.

```
public interface com.ibm.OS4690.DeviceManagerInterface
{
    //Methods
    *public void setIOPHandler(IOPHandler iop);
    *public void setExtendedIOPHandler(ExtendedIOPHandler iop);
    *public void setANDisplay1Handler(ANDisplayHandler display);
    *public void setANDisplay2Handler(ANDisplayHandler display);
    *public void setCashReceiptMonitor(CashReceiptMonitor cr);
```

```

*public void setMSRHandler(MSRHandler msr);
*public void setPOSPrinterHandler(POSPrinterHandler prn);
*public void setDeviceRegistrationComplete();
*public void IOPDataAvailable();
*public void MSRDataAvailable();
*public void POSPrinterDataAvailable(int nStation);
public void addDMDeviceRegistrationCompleteListener(DMDeviceRegistrationCompleteListener listener);
public void setJavaInvocationHandler(JavaInvocationHandler jih);
}

```

\* Functions in the same manner as a method with the same signature in class DeviceManager. See “Class DeviceManager” on page 540 for a description.

## DeviceManagerInterface Methods

This section describes the methods for the DeviceManagerInterface class.

### addDMDeviceRegistrationCompleteListener()

This method registers a listener that is notified when the corresponding DeviceManager has completed its handler registration.

```
public void addDMDeviceRegistrationCompleteListener(DMDeviceRegistrationCompleteListener listener);
```

### setJavaInvocationHandler()

This method registers a handler for Java method invocation requests from a native application. By default, an implementation is registered that performs normal static method invocation. A new handler implementation might be needed in such cases as the multiple emulated terminal environment.

```
public void setJavaInvocationHandler(JavaInvocationHandler jih);
```

---

## Interface DeviceManagerInterfaceMultiJVM

This method is a DeviceManagerInterface extension representing a DeviceManager running in a true terminal environment. An implementation of this interface can be obtained through DeviceManager.singleton(). Communication with the native application is not initiated until the DeviceManager in the primary JVM calls setDeviceRegistrationComplete(). The DMDeviceRegistrationCompleteListener registered through addDMDeviceRegistrationCompleteListener() is notified of registration completion of other DeviceManager's in the terminal. An application running in the primary JVM can take advantage of this behavior to delay its call to setDeviceRegistrationComplete() until it has been notified of device registration completion from the secondary JVM's it is expecting to run. This prevents the native application from communicating with a DeviceManager that is still initializing. By default, a DeviceManager can be used only in one JVM, which must be the primary JVM. To allow DeviceManagers to run in a secondary JVM, a call must be made in the primary JVM to enableMultiJVM().

**Note:** A DeviceManager in a secondary JVM is allowed only to register handlers for ANDisplay2 and method invocation.

```

public interface com.ibm.OS4690.DeviceManagerInterfaceMultiJVM extends DeviceManagerInterface
{
    //Methods
    public void enableMultiJVM() throws java.rmi.RemoteException
}

```

## DeviceManagerInterfaceMultiJVM Methods

This section describes the methods for the DeviceManagerInterfaceMultiJVM class.

### enableMultiJVM()

This method allows DeviceManagers in secondary JVM's to be used.

```
public void enableMultiJVM() throws java.rmi.RemoteException
```

---

## Interface DeviceManagerInterfaceTSS

This method is a DeviceManagerInterface extension representing a DeviceManager running in an environment with multiple emulated terminals. An implementation of this class can be obtained through DeviceManager.createDeviceManager(int termID).

```
public interface com.ibm.OS4690.DeviceManagerInterfaceTSS extends DeviceManagerInterface
{
    //Methods
    public void dispose()
}
```

### DeviceManagerInterfaceTSS Methods

This section describes the methods for the DeviceManagerInterfaceTSS class.

#### dispose()

This method disposes the DeviceManager associated with this DeviceManagerInterfaceTSS.

```
public void dispose()
```

---

## Interface DMDeviceRegistrationCompleteListener

This method is an interface for notification of registration completion of a DeviceManager. See DeviceManagerInterface.addDMDeviceRegistrationCompleteListener().

In a true terminal environment, notification of registration is provided for DeviceManagers in other JVM's. The value of deviceManagerID is the ID of the JVM containing the DeviceManager.

In an environment with multiple emulated terminals, the listener is informed only of registration completion for DeviceManagers that it explicitly registers interest in. The value of deviceManagerID is the emulated terminal number associated with the DeviceManager.

```
public interface com.ibm.OS4690.DMDeviceRegistrationCompleteListener
{
    //Methods
    public void handleDeviceRegistrationComplete(int deviceManagerID);
}
```

### DMDeviceRegistrationCompleteListener Methods

This section describes the methods for the DMDeviceRegistrationCompleteListener class.

#### dispose()

This method disposes the DeviceManager that is associated with this DeviceManagerInterfaceTSS.

```
public void dispose()
```

---

## Interface JavaInvocationHandler

This method is an interface for performing method invocation requests from a native application. A class implementing this interface can be registered through DeviceManagerInterface.setJavaInvocationHandler().

```
public interface com.ibm.OS4690.JavaInvocationHandler
{
    //Methods
    public JavaInvocationResult invokeMethod(String className, String methodName, Object[] args);
}
```



## JavaInvocationHandler Methods

This section describes the methods for the `JavaInvocationHandler` class.

### invokeMethod()

This method performs method invocation for a native application request.

Parameters:

**className**

Fully-qualified class name (for example, `com.xyz.Main`)

**methodName**

Name of method (for example, `dispose`)

**args** Arguments for the invocation. The length of `args` is  $\geq 0$ . Each element of the array is one of the following:

- `java.lang.Byte`
- `java.lang.Short`
- `java.lang.Integer`
- `java.lang.String`

```
public JavaInvocationResult invokeMethod(String className, String methodName, Object[] args);
```

---

## Class JavaInvocationResult

This method is a class representing the result of a method invocation. See `JavaInvocationHandler.invokeMethod()`.

```
public class com.ibm.OS4690.JavaInvocationResult
{
    //Methods
    public JavaInvocationResult(Object returnValue);
    public JavaInvocationResult(Throwable exception);
}
```

### JavaInvocationResult Constructors

This section describes the constructors for the `JavaInvocationResult` class.

#### JavaInvocationResult(Object returnValue)

This method constructs a `JavaInvocationResult` for a method that successfully executed. `returnValue` should be one of the following values:

- `java.lang.Byte`
- `java.lang.Short`
- `java.lang.Integer`
- `java.lang.String`
- `null`

The value of `returnValue` should be `null` if the method being invoked has a return type of `void`.

#### JavaInvocationResult(Throwable exception)

This method constructs a `JavaInvocationResult` for a method that threw an exception. The native application is given a character string containing the class name of exception followed by the string returned by `exception.getMessage()`.



---

## Class TerminalKeyboardLights

Java terminal applications, which are intended to be run in a touch screen environment without a physical keyboard, can query native terminal services to determine whether or not the Wait, Offline, or MsgPend lights should be on. Methods are also provided to retrieve text associated with a light that is turned on. Special consideration might be necessary when using this function together with JavaPOS drivers for the POS keyboard. In particular, the Wait light is generally controlled by the terminal sales application. If a Java application is directly controlling the Wait light through JavaPOS, it is likely that the status returned from these calls might not accurately reflect the status of a physical keyboard light. This situation occurs when a Java application directly turns on the Wait light through JavaPOS because the native OS terminal services is not updated.

```
public class com.ibm.OS4690.TerminalKeyboardLights
{
    // There are no public constructors

    // Methods
    // All of these methods are static and may throw com.ibm.OS4690.FlexosException
    public static boolean isWaitLightOn();
    public static boolean isOfflineLightOn();
    public static boolean isMsgPendLightOn();
    public static String readWaitMessage();
    public static String readOfflineMessage();
    public static String readMsgPendMessage();
}
```

## TerminalKeyboardLights Methods

This section describes the methods for the TerminalKeyboardLights class.

### isWaitLightOn()

This method returns true if the terminal services status indicates the Wait light should be turned on.

```
public static boolean isWaitLightOn() throws com.ibm.OS4690.FlexosException
```

This method throws a FlexosException if an IO error occurs while communicating with the native terminal services driver.

### isOfflineLightOn()

This method returns true if the terminal services status indicates the Offline light should be turned on.

```
public static boolean isOfflineLightOn() throws com.ibm.OS4690.FlexosException
```

This method throws a FlexosException if an IO error occurs while communicating with the native terminal services driver.

### isMsgPendLightOn()

This method returns true if the terminal services status indicates the MsgPend light should be turned on.

```
public static boolean isMsgPendLightOn() throws com.ibm.OS4690.FlexosException
```

This method throws a FlexosException if an IO error occurs while communicating with the native terminal services driver.

### readWaitMessage()

This method returns a String containing the text associated with the Wait light. This text is the same text that would have been displayed if a physical keyboard was attached and the operator had pressed the **S1-1-S2** system key sequence to display the Wait message.

```
public static java.lang.String readWaitMessage()
    throws com.ibm.OS4690.FlexosException
```

This method returns null if there is no Wait message available. This method throws a `com.ibm.OS4690.FlexosException` if an IO error occurs while communicating with the native terminal services driver.

### **readOfflineMessage()**

This method returns a `String` containing the text associated with the Offline light. This text is the same text that would have been displayed if a physical keyboard was attached and the operator had pressed the **S1-2-S2** system key sequence to display the Offline message.

```
public static java.lang.String readOfflineMessage()  
throws com.ibm.OS4690.FlexosException
```

This method returns null if there is no Offline message available. This method throws a `com.ibm.OS4690.FlexosException` if an IO error occurs while communicating with the native terminal services driver.

### **readMsgPendMessage()**

This method returns a `String` containing the text associated with the MsgPend light. This text is the same text that would have been displayed if a physical keyboard was attached and the operator had pressed the **S1-3-S2** system key sequence to display the MsgPend message.

```
public static java.lang.String readMsgPendMessage()  
throws com.ibm.OS4690.FlexosException
```

This method returns null if there is no MsgPend message available. This method throws a `com.ibm.OS4690.FlexosException` if an IO error occurs while communicating with the native terminal services driver.

---

## **Class ANDisplayHandler**

The `ANDisplayHandler` class is an interface that allows you to implement `ANDisplay1`, `ANDisplay2`, or both. This interface resides in `OS4690.ZIP`. When implementing an `ANDisplayHandler`, you must emulate exactly the types of responses that would have been returned by the replaced hardware.

If the video is configured as the system display, you can configure and redirect the `ANDisplayHandler` without having a 2x20 display physically attached to the terminal. This would eliminate one or two pieces of hardware that typically would not be used on a GUI implementation.

The `ANDisplayHandler` only receives display requests from the CBASIC or C application. It does not receive state table prompts, messages, and keystrokes from the Java IO Processor. You must implement a `PromptChangeListener` (see “Class `PromptChangeListener`” on page 605) in order to receive prompts, messages and keystrokes from the Java IO Processor.

If the terminal load definition has been configured to use display redirection from either `ANDISPLAY` or `ANDISPLAY2`, the CBASIC or C application must issue its display request to the redirected device instead even if the system display is configured to be a different display device such as the `VDISPLAY`. The application should check the terminal application status to determine if redirection is being used by testing byte 59. (See Table 33 on page 297 for more information.) If redirection is being used for either `ANDISPLAY` or `ANDISPLAY2`, the application should make sure that it opens the appropriate display. The code below is an example of how an application should test to see if the `ANDISPLAY` is being redirected. If it is being redirected, the application opens the appropriate display.

```
I=4  
CALL ADXSERVE (TS.DEVSTAT.I.J.TS.STAT$)! get status data  
REDIRECT = ASC(MID$(TS.STAT$.59.1))    ! obtain redirection byte  
  
IF (REDIRECT AND 40H) THEN BEGIN      ! If ANDISPLAY is being redirected
```

```

OPEN "ANDISPLAY:" AS 19      ! open this display. The application
                             ! will write prompts and error message
                             ! to it.
ENDIF

```

When using display redirection and an enhanced full-screen state table, the `ANDisplayHandler` must be used to receive display requests from the CBASIC or C application even though the CBASIC or C application might be a full-screen application designed to be used with a video display. This means that the full-screen application must detect that redirection is being used and must open and issue requests to the redirected display device. Because the `ANDISPLAY` and `ANDISPLAY2` are the only display devices that can be redirected, the full-screen application which uses redirection must make sure that it does not issue writes that exceed 40 bytes in order to avoid a run-time error. In this case where a full-screen application is being used, the `ANDisplayHandler` should implement the same number of rows and columns as defined for the video display, rather than implementing two rows by twenty columns.

This interface documents methods for communicating between the Java GUI and the `ANDisplay`.

Described below are the methods for communicating between the Java GUI and the `ANDisplay`.

## ANDisplayHandler Methods

### **open()**

This method requests the `ANDisplayHandler` class to perform the open. If it is unable to open the device, it throws `ANDisplayHandlerException`.

```
public abstract void open() throws ANDisplayHandlerException
```

### **close()**

This method requests the `ANDisplayHandler` class to perform the close.

```
public abstract void close()
```

### **clear()**

This method requests the `ANDisplayHandler` class to perform the clear. If it is unable to perform the clear, it throws `ANDisplayHandlerException`.

```
public abstract void clear() throws ANDisplayHandlerException
```

### **locate()**

This method requests the `ANDisplayHandler` class to move the cursor to the specified row and column. It throws `ANDisplayHandlerException` when row or column values specified as parameters are out of bounds.

```
public abstract void locate(short row, short column) throws
ANDisplayHandlerException
```

### **Parameters:**

**row**     The row location. Valid values are 1 and 2.

### **column**

The column location. Valid values are from 1 to 20.

### **read()**

This method requests the `ANDisplayHandler` class to perform a read. It throws `ANDisplayHandlerException` when it is unable to perform the read.

```
public abstract int read(byte buf[]) throws ANDisplayHandlerException
```

### **Parameters:**

**buf**     The buffer to contain the data being read.

**Returns:** The method returns the number of bytes read.

### **write()**

This method requests the `ANDisplayHandler` class to perform the write. It throws `ANDisplayHandlerException` exception if it is unable to perform the write.

```
public abstract int write(byte buf[]) throws ANDisplayHandlerException
```

#### **Parameters:**

**buf**     The buffer containing the data to write.

**Returns:** The method returns the number of bytes written.

### **getRow()**

This method queries the `ANDisplayHandler` class for its current row location.

```
public abstract short getRow()
```

**Returns:** This method returns an integer, either 1 or 2 for row 1 or 2.

### **getColumn()**

This method queries the `ANDisplayHandler` class for its current column location.

```
public abstract short getColumn()
```

**Returns:** This method returns an integer from 1 to 20 indicating column 1 to 20.

### **getDisplayType()**

This method queries the `ANDisplayHandler` class for its display type.

```
public abstract short getDisplayType()
```

**Returns:** This method returns the display type.

---

## **Class IOHandler**

The `IOHandler` class is an interface that allows you to implement the full capability of the `IOProcessor` in Java. It resides in `OS4690.ZIP`.

The `IOHandler` class is one of the most complex devices in the system. To help you develop unique GUIs to support your applications, there is an implementation of the `IOHandler` class called the `JIOP`. We highly recommend that you use this implementation for your `IOHandler` class, because it is designed to handle all of the unique functions and responsibilities of the `IOProcessor`. For additional information on `JIOP`, see “Java I/O Processor Functions” on page 586.

Documented below are the methods for communicating between the Java GUI and the `IOHandler`.

## **IOHandler Methods**

### **close()**

This method requests the `IOHandler` to perform the close.

```
public abstract void close()
```

### **configureBuffers(short numberOfBuffers, int bufferSize)**

This method configures the number and size of buffers the `IOHandler` should use. (The operating system uses two bytes for each buffer.)

```
public abstract void configureBuffers(short numberOfBuffers,int bufferSize)
```

**Parameters:**

**numberOfBuffers — (1–10)**

Number of input sequences to buffer

**bufferSize — (16–32767)**

Maximum size for each buffer

**getLastStateRead()**

This method returns the last state of the IO processor.

```
public abstract short getLastStateRead()
```

**Returns:** This method returns the last state of the device.

**isDeviceInPriorityMode()**

This method determines if the device is in Priority mode.

```
public abstract boolean isDeviceInPriorityMode()
```

**isDeviceLocked()**

This method determines if a lock has been issued to the device.

```
public abstract boolean isDeviceLocked()
```

**isDevicePurged()**

This method determines if a purge has been issued to the device.

```
public abstract boolean isDevicePurged()
```

**isIOPDataAvailable()**

This method queries the IOHandler to see if there is IO processor data available to be read.

```
public abstract boolean isIOPDataAvailable()
```

**loadFormatTable(byte[] tableName)**

This method requests the IOHandler to load the format table into memory. The method throws IOException if it is unable to load the format table.

```
public abstract void loadFormatTable(byte[] tableName) throws IOException
```

**Parameters:**

**tableName**

Format table name

**loadInputStateTable(byte[] tableName)**

This method requests the IOHandler to load the input sequence table into memory. It throws IOException if it is unable to load the input state table.

```
public abstract void loadInputStateTable(byte[] tableName) throws
IOException
```

**Parameters:**

**tableName**

Input sequence table name

**loadModuloTable(byte[] tableName)**

This method requests the IOHandler to load the modulo table into memory. It throws IOException if it is unable to load the modulo table.

```
public abstract void loadModuloTable(byte[] tableName) throws
IOException
```

**Parameters:**

**tableName**

Modulo table name

**lockDevice(boolean purge)**

This method requests the IOPHandler to perform the lock. When locked, the IO processor rejects all keyboard, reader, and scanner input. If you specify purge as true, all data queued to your application is discarded.

```
public abstract void lockDevice(boolean purge)
```

**Parameters:**

**purge**

**false** Do not discard data.

**true** Discard data queued to application.

**open()**

This method requests the IOPHandler to perform the open. It throws IOPHandlerException if it is unable to perform the open.

```
public abstract void open() throws IOPHandlerException
```

**read(byte[] buf)**

This method requests the IOPHandler to perform the read. It throws IOPHandler if it is unable to perform the read.

```
public abstract int read(byte[] buf) throws IOPHandlerException
```

**Parameters:**

**buf** Buffer to contain data.

**Returns:** This method returns the number of bytes read.

**setKeyboardData(String keyboardFileName, short index)**

Sends the IOP the name of the current keyboard translate table file, and the index into the file to the record being currently used.

```
public abstract void setKeyboardData(String keyboardFileName, short index)
```

**Parameters:**

**keyboardFileName**

The name of the keyboard translate table file currently in use.

**index** The index into the keyboard translate table file to the translate table currently being used.

**unlock()**

This method requests the IOPHandler to perform the unlock. It throws IOPHandlerException if it is unable to unlock.

```
public abstract void unlock() throws IOPHandlerException
```

**unlock(short state)**

This method requests the IOPHandler to perform the unlock. It throws IOPHandlerException if it is unable to unlock to the specified state.

```
public abstract void unlock(short state) throws IOPHandlerException
```

**Parameters:**

**state** The new state you want to unlock to.

### **unlock(short state, boolean priority)**

This method requests the IOPHandler to perform the unlock. It throws IOPHandlerException if it is unable to unlock to the specified state..

```
public abstract void unlock(short state,boolean priority) throws IOPHandlerException
```

#### **Parameters:**

**state** The new state to unlock to.

#### **priority**

This causes the next data entered to be read before the data in the IO processor queue.

---

## **Class ExtendedIOPHandler**

This class is an interface that extends the IOPHandler class. In addition to the methods previously defined in the IOPHandler class, this class provides three new methods as described below. The existing Java applications that implemented the IOPHandler interface can continue to function normally. However, in order to make use of the new methods, you must implement the ExtendedIOPHandler interface.

### **ExtendedIOPHandler Methods**

The ExtendedIOPHandler Methods class contains the following methods:

#### **write(byte[] buf)**

This method requests the ExtendedIOPHandler to perform a write. It throws IOPHandler Exception if it is unable to perform the write.

```
public abstract int write(byte[] buf) throws IOPHandlerException
```

#### **Parameters:**

**buf** Buffer that contains the data to be written.

**Returns:** This method returns the number of bytes written.

#### **getStatus()**

This method requests the ExtendedIOPHandler to return the current status.

```
public abstract int getStatus()
```

**Returns:** This method returns the current status as a 4-byte integer. The format of this integer is FTTTSLSH as described in the *4680 Basic Language Reference*. By returning -1 you can let the Device Manager construct the 4 status bytes. Otherwise, the value returned by the implementation is used. If the returned value is -1, the Device Manager constructs the 4 status bytes as it was done in the IOPHandler interface. The following 4 methods are used as appropriate to construct the 4 status bytes:

```
isDeviceLocked()  
isDevicePurged()  
isDeviceInPriorityMode()  
getLastStateRead()
```

This is implemented this way so as to give the user more flexibility in terms of returning the appropriate status information to the C or CBASIC application.

#### **unlock(sort sState, byte byTT, byte byFF)**

This method requests the ExtendedIOPHandler to unlock to a given state and tab order.

```
public abstract void unlock(short sState, byte byTT, byte byFF) throws  
IOPHandlerException
```

#### **Parameters:**

**sState** The state to which to unlock to.

**byTT** The byte containing the tab order.

**byFF** The byte with status information. See details on the PUTLONG statement in the *4680 Basic Language Reference* for additional details on this byte.

---

## Class CashReceiptMonitor

The CashReceiptMonitor class is an interface that allows the Java GUI to receive a copy of the data going to the printer's cash receipt station. It resides in OS4690.ZIP.

### CashReceiptMonitor Method

The CashReceiptMonitor class contains one method.

#### printData()

This method gives the CashReceiptMonitor a copy of the data sent to the printer.

```
public abstract void printData(byte[] buf)
```

#### Parameters:

**buf** Identifies the printer's data buffer.

---

## Class MSRHandler

The MSRHandler class is an interface that allows you to implement the full capability of the MSR in Java. It resides in OS4690.ZIP.

**Note:** See the *4690 OS: Programming Guide* for the format of MSR data to be returned to the legacy application. This varies according to the number of tracks in the MSR and which tracks are configured to be read.

### MSRHandler Methods

The MSRHandler class contains the following methods:

#### open()

This method requests the MSRHandler to perform an open. It throws an MSRHandlerException if it is unable to perform the open.

```
public abstract void open() throws MSRHandlerException
```

#### close()

This method requests the MSRHandler to perform a close.

```
public abstract void close()
```

#### read(byte[] buf)

This method requests the MSRHandler to perform the read. It throws an MSRHandlerException if it is unable to perform the read.

```
public abstract int read(byte[] buf) throws MSRHandlerException
```

#### Parameters:

**buf** Buffer to contain data.

**Returns:** This method returns the number of bytes read.

#### lock(boolean bLock)

This method requests the MSRHandler to perform a lock.

```
public abstract void lock(boolean bLock)
```



**Parameters:**

**bLock** If true, MSR should be locked, if not true MSR should be unlocked.

**getStatus()**

This method requests the MSRHandler to return the current status of the device.

```
public abstract int getStatus()
```

**Returns:** This method returns the current status as a 4-byte integer. See the *4680 Basic Language Reference* for the format of this integer.

**isDeviceLocked()**

This method returns the current locked status of the MSR.

```
public abstract boolean isDeviceLocked()
```

**Returns:** True if the MSR is locked, otherwise returns false.

**isMSRDataAvailable()**

This method queries the MSRHandler to see if there is MSR data available to read.

```
public abstract boolean isMSRDataAvailable()
```

**Returns:** True if there is data available to be read, otherwise returns false.

---

## Class POSPrinterHandler

The POSPrinterHandler class is an interface that allows you to implement the full capability of the POSPrinter in Java. It resides in OS4690.ZIP.

### POSPrinterHandler Methods

In the following methods, the argument nStation identifies a specific printer station: Cash Receipt, Document Insert, or Summary Journal. The POSPrinterHandler class contains the following methods:

**open(int nStation)**

This method requests the POSPrinterHandler to perform an open. It throws a POSPrinterHandlerException if it is unable to perform the open.

```
public abstract void open(int nStation) throws POSPrinterHandlerException
```

**Parameters:**

**nStation**

This parameter can be one of the following:

- DeviceManagerConst.PS\_CASH\_RECEIPT
- DeviceManagerConst.PS\_DOCUMENT\_INSERT
- DeviceManagerConst.PS\_SUMMARY\_JOURNAL

**close(int nStation, boolean bTClose)**

This method requests the POSPrinterHandler to perform the close.

**Note:** A legacy (C or CBASIC) application should issue a TCLOSE before closing a printer station to ensure that all print data is sent to the printer.

```
public abstract void close(int nStation, boolean bTClose))
```

**Parameters:**

**nStation**

This parameter can be one of the following:

- DeviceManagerConst.PS\_CASH\_RECEIPT
- DeviceManagerConst.PS\_DOCUMENT\_INSERT
- DeviceManagerConst.PS\_SUMMARY\_JOURNAL

#### **bTClose**

If true, this parameter requests a TCLOSE, otherwise it performs a normal CLOSE.

#### **write(int nStation, byte[] buf)**

This method requests the POSPrinterHandler to perform a write. It throws a POSPrinterHandlerException if it is unable to perform the write.

```
public abstract int write(int nStation, byte[]buf)throws POSPrinterHandlerException
```

#### **Parameters:**

##### **nStation**

This parameter can be one of the following:

- DeviceManagerConst.PS\_CASH\_RECEIPT
- DeviceManagerConst.PS\_DOCUMENT\_INSERT
- DeviceManagerConst.PS\_SUMMARY\_JOURNAL

**buf** This parameter is the buffer that contains the data to be written.

**Returns:** This method returns the number of bytes written.

#### **read(int nStation, byte[] buf)**

This method requests the POSPrinterHandler to perform a read. It throws a POSPrinterHandlerException if it is unable to perform the read.

```
public abstract int read(int nStation, byte[]buf)throws POSPrinterHandlerException
```

#### **Parameters:**

##### **nStation**

This parameter can be one of the following:

- DeviceManagerConst.PS\_CASH\_RECEIPT
- DeviceManagerConst.PS\_DOCUMENT\_INSERT

**buf** This parameter is the buffer that contains the data to be read.

**Returns:** This method returns the number of bytes read.

#### **getStatus(int nStation)**

This method requests the POSPrinterHandler to return the current status.

```
public abstract int getStatus(int nStation)
```

#### **Parameters:**

##### **nStation**

This parameter can be one of the following:

- DeviceManagerConst.PS\_CASH\_RECEIPT
- DeviceManagerConst.PS\_DOCUMENT\_INSERT
- DeviceManagerConst.PS\_SUMMARY\_JOURNAL

**Returns:** This method returns the current status as a 4-byte integer. See “Determining the Printer Status” on page 124 for more information.

**Note:** BASIC GETLONG and PUTLONG pass 4-byte integers between BASIC and Java applications. The bytes in this integer are reversed in transit. If a 4690 BASIC application performs a PUTLONG to a

non-4610 printer or to the Summary Journal station of a 4610 printer, the high byte is discarded and the remaining three bytes are reversed. For example, if the BASIC application does a PUTLONG of the value 12345678H, the Java application receives the value 0x785634. Other instances of PUTLONG or GETLONG to a printer station pass all four bytes that are received in reverse order, that is, the value 12345678H is received as 0x78563412.

### **setStatus(int nStation, int nStatus)**

This method requests the POSPrinterHandler to set the current status to the new value. This method returns the current status as a 4-byte integer. The format of this integer is FFTTSLSH as described in the *4690 OS: User's Guide*.

```
public abstract void setStatus(int nStation, int nStatus)
```

#### **Parameters:**

##### **nStation**

This parameter can be one of the following:

- DeviceManagerConst.PS\_CASH\_RECEIPT
- DeviceManagerConst.PS\_DOCUMENT\_INSERT
- DeviceManagerConst.PS\_SUMMARY\_JOURNAL

##### **nStatus**

This parameter is a 4-byte integer containing the 4 status bytes.

### **isMICRDataAvailable()**

This method queries the POSPrinterHandler to see if there is POSPrinter data available to be read.

```
public abstract boolean isMICRDataAvailable()
```

---

## **Errors and Exceptions**

To handle error conditions, 4690 OS provides you with classes that set a 4690 return code that CBASIC recognizes for that device. These classes are ANDisplayHandlerException, IOPHandlerException, MSRHandlerException, and POSPrinterHandlerException. You can use each of these to flag an error condition in Java, and to inform the C or CBASIC application in a way consistent with its other errors.

### **Class ANDisplayHandlerException**

This class provides the 4690 return codes that the 2x20 displays would have returned based on the input sent in.

This interface documents methods for handling exceptions when communicating between the Java GUI and the ANDisplay.

#### **ANDisplayHandlerException Constructors**

This section describes constructors for the ANDisplayHandlerException class.

##### **ANDisplayHandlerException():**

```
ANDisplayHandlerException(int retcode)
```

##### **Parameters:**

##### **retcode**

The ANDisplayHandlerException return code.

##### **ANDisplayHandlerException(String):**

```
ANDisplayHandlerException(String detailString)
```

*Parameters:*

**detailString**

The detailed message is returned.

## **ANDisplayHandlerException Methods**

***getMessage():***

```
public String getMessage()
```

*Returns:* This method returns the text message of the Exception.

*Overrides:* This method overrides the `getMessage()` method in class `Throwable`.

***getReturnCode():***

```
public int getReturnCode()
```

*Returns:* This method returns the 4690 return code for the `ANDisplay`.

***setDeviceOfflineRC():*** This method sets the return code for the `ANDisplay` stating that the device is offline.

```
public void setDeviceOfflineRC()
```

***setInvalidCursorPositionRC():*** This method sets the return code for the `ANDisplay` stating that an invalid cursor position was specified.

```
public void setInvalidCursorPositionRC()
```

***setHardwareFailureRC():*** This method sets the return code for the `ANDisplay` stating that a hardware error occurred.

```
public void setHardwareFailureRC()
```

## **Class IOHandlerException**

This class provides the 4690 OS return codes that the `IOProcessor` would have returned based on input sent in. This class has been used by our `JIOP` to handle error conditions it encounters.

This interface documents the methods for processing exceptions when communicating between the Java GUI and the `IOHandler`.

## **IOHandlerException Constructors**

***IOHandlerException():***

```
public IOHandlerException()
```

***IOHandlerException(String msg):***

```
public IOHandlerException(String msg)
```

## **IOHandlerException Methods**

***getMessage():*** This method returns the text message of the Exception.

```
public String getMessage()
```

*Returns:* This method returns the text message of the Exception.

*Overrides:* This method overrides the `getMessage` method in class `Throwable`.

***getReturnCode():*** This method returns the 4690 return code for the `IOProcessor`.

```
public int getReturnCode()
```

**Returns:** This method returns the 4690 Return code for the IOProcessor.

**setAnposKbBufferOverrunRC():** This method sets the return code when an alphanumeric or ANPOS keyboard buffer overrun has occurred.

```
public void setAnposKbBufferOverrunRC()
```

**setAnposKbFailureRC():** This method sets the return code when an alphanumeric or ANPOS keyboard failure has occurred.

```
public void setAnposKbFailureRC()
```

**setAnposKbOfflineRC():** This method sets the return code when an alphanumeric or ANPOS keyboard is offline.

```
public void setAnposKbOfflineRC()
```

**setBufferOverFlowRC():** This method sets the return code for a buffer overflow that occurs when the amount of data being returned from the READ exceeds the caller's buffer size.

```
public void setBufferOverFlowRC()
```

**setDefaultRC():** This method sets the return code when an unknown exception has occurred.

```
public void setDefaultRC()
```

**setDriverLockedRC():** This method sets a return code when a read was requested but the driver is currently locked.

```
public void setDriverLockedRC()
```

**setInputStateTableNotLoadedRC():** This method sets a return code when there is a request for an Input State Table that has not yet been loaded.

```
public void setInputStateTableNotLoadedRC()
```

**setIOPOpenErrorRC():** This method sets a return code when there is a request to open the IOProcessor when the IOProcessor has already been opened.

```
public void setIOPOpenErrorRC()
```

**setKBBufferOverrunRC():** This method sets the return code when a keyboard buffer overrun has occurred.

```
public void setKBBufferOverrunRC()
```

**setKBFailureRC():** This method sets the return code when a keyboard failure has occurred.

```
public void setKBFailureRC()
```

**setKBOfflineRC():** This method sets the return code when a keyboard is offline.

```
public void setKBOfflineRC()
```

**setKBOrDisplayOfflineRC():** This method sets the return code when a keyboard or display is offline.

```
public void setKBOrDisplayOfflineRC()
```

**setMatrixKbFailureRC():** This method sets the return code when a matrix keyboard failure has occurred.

```
public void setMatrixKbFailureRC()
```

**setMatrixKbOfflineRC():** This method sets the return code when a matrix keyboard is offline.

```
public void setMatrixKbOfflineRC()
```

***setMatrixKbBufferOverrunRC():*** This method sets the return code when a matrix keyboard buffer overrun error has occurred.

```
public void setMatrixKbbufferOverrunRC()
```

***setNoResourcesRC():*** This method sets the return code when there is insufficient system storage for the input buffers.

```
public void setNoResourcesRC()
```

***setOCRBufferOverrunRC():*** This method sets the return code when a magnetic wand or OCR device buffer overrun has occurred.

```
public void setOCRBufferOverrunRC()
```

***setOCRFailureRC():*** This method sets the return code when a magnetic wand or the OCR device is failing.

```
public void setOCRFailureRC()
```

***setOCROfflineRC():*** This method sets the return code when a magnetic wand or the OCR device is offline.

```
public void setOCROfflineRC()
```

***setScannerBufferOverrunRC():*** This method sets the return code when a scanner buffer overrun has occurred.

```
public void setScannerBufferOverrunRC()
```

***setScannerFailureRC():*** This method sets the return code when a scanner is failing.

```
public void setScannerFailureRC()
```

***setScannerOfflineRC():*** This method sets the return code when a scanner is offline.

```
public void setScannerOfflineRC()
```

***setStateNotValidRC():*** This method sets a return code when an unlock method specifies an invalid IOProcessor state.

```
public void setStateNotValidRC()
```

***setTableNotFoundRC():*** This method sets a return code when there is a request for an IOProcessor state table that cannot be found.

```
public void setTableNotFoundRC()
```

***setWriteBufferNotValidRC():*** This method sets the return code when the user's buffer provided for the write method is invalid.

```
public void setWriteBufferNotValidRC()
```

## Class MSRHandlerException

This class provides the 4690 OS return codes that the MSR would have returned based on input sent in.

This interface documents the methods for processing exceptions when communicating between the Java GUI and the MSRHandler.

### MSRHandlerException Constructors

***MSRHandlerException():***

```
public MSRHandlerException()
```

The default MSRHandler constructor.

***MSRHandlerException(String msg):***

```
public MSRHandlerException(String msg)
```

Another MSRHandler constructor.

*Parameters:*

**msg** Detailed information about the MSRHandler exception.

## **MSRHandlerException Methods**

***getMessage():*** This method returns the text message of the Exception.

```
public String getMessage()
```

*Returns:* This method returns detailed description of the Exception.

*Overrides:* This method overrides the getMessage method in class Throwable.

***getReturnCode():*** This method returns the 4690 return code for the MSRHandler.

```
public int getReturnCode()
```

*Returns:* This method returns the 4690 Return code for the MSRHandler.

***setDeviceOfflineRC():*** This method sets the return code for the MSRHandler stating that the MSR is offline.

```
public void setDeviceOfflineRC()
```

***setDeviceNotAttachedRC():*** This method sets the return code for the MSRHandler stating that the MSR driver is locked or not responding.

```
public void setDeviceNotAttachedRC()
```

***setUnableToReadRC():*** This method sets the return code for the MSRHandler stating that the read attempt failed.

```
public void setUnableToReadRC()
```

***setDevNotAttachedRC():*** This method sets the return code for MSRHandler stating that the MSR is not attached.

```
public void setDevNotAttachedRC()
```

***setKBOOrDualTrkNotAttachedRC():*** This method sets the return code for the MSRHandler stating that the keyboard or the Dual-track MSR is not attached.

```
public void setKBOOrDualTrkNotAttachedRC()
```

***setDataNotCompleteRC():*** This method sets a return code for the MSRHandler stating that the data received is not complete.

```
public void setDataNotCompleteRC()
```

***setDevAlreadyOpenedRC():*** This method sets a return code for the MSRHandler stating that the MSR is already opened by this application.

```
public void setDevAlreadyOpenedRC()
```

***setBufferTooSmallRC():*** This method sets a return code for the MSRHandler stating that the buffer is too small.

```
public void setBufferTooSmallRC()
```

## Class POSPrinterHandlerException

This class provides the 4690 OS return codes that the POSPrinter would have returned based on input sent in.

This interface documents the methods for processing exceptions when communicating between the Java GUI and the POSPrinterHandler.

### POSPrinterHandlerException Constructors

#### ***POSPrinterHandlerException():***

```
public POSPrinterHandlerException()
```

The default POSPrinterHandler constructor.

#### ***POSPrinterHandlerException(String msg):***

```
public POSPrinterHandlerException(String msg)
```

Another POSPrinterHandler constructor.

#### *Parameters:*

**msg** Detailed information about the POSPrinterHandler exception.

### POSPrinterHandlerException Methods

***getMessage():*** This method returns the text message of the Exception.

```
public String getMessage()
```

*Returns:* This method returns the detailed description of the Exception.

*Overrides:* This method overrides the getMessage method in the class Throwable.

***getReturnCode():*** This method returns the 4690 return code for the POSPrinterHandler.

```
public int getReturnCode()
```

*Returns:* This method returns the 4690 Return code for the POSPrinterHandler.

***setDeviceOfflineRC():*** This method sets the return code for the POSPrinterHandler stating that the printer is offline.

```
public void setDeviceOfflineRC()
```

***setReceivedIllegalDataRC():*** This method sets the return code for the POSPrinterHandler stating that the data received is invalid.

```
public void setReceivedIllegalDataRC()
```

***setInvalidOperationRC():*** This method sets the return code for the POSPrinterHandler stating that the method requested is not supported on the printer.

```
public void setInvalidOperationRC()
```

***setBufferOverflowRC():*** This method sets the return code for POSPrinterHandler stating that the buffer size requested is more than the maximum supported.

```
public void setBufferOverflowRC()
```

***setDefaultRC():*** This method sets the return code for the POSPrinterHandler stating that some unknown exception occurred.

```
public void setDefaultRC()
```



**setNoMicrRC():** This method sets a return code for the POSPrinterHandler stating that there is no MICR on this printer.

```
public void setNoMicrRC()
```

**setCmdRejectRC():** This method sets a return code for the POSPrinterHandler stating that the printer has returned a command reject.

```
public void setCmdRejectRC()
```

**setDeviceNotAttachedRC():** This method sets a return code for the POSPrinterHandler stating that the selected device is not attached to this printer.

```
public void setDeviceNotAttachedRC()
```

**setTimedOutOnReadRC():** This method sets a return code for the POSPrinterHandler stating that the read command timed out.

```
public void setTimedOutOnReadRC()
```

**setDeviceFailureRC():** This method sets a return code for the POSPrinterHandler stating that the device is about to fail. This could be a head movement error.

```
public void setDeviceFailureRC()
```

**setCoverOpenErrorRC():** This method sets a return code for the POSPrinterHandler stating that the cover is open and the print line is in error.

```
public void setCoverOpenErrorRC()
```

**setWriteLogoNotSupportedRC():** This method sets a return code for the POSPrinterHandler stating that the WRITE LOGO method is not supported on this printer.

```
public void setWriteLogoNotSupportedRC()
```

**setJrnPaperErrRC():** This method sets a return code for the POSPrinterHandler stating that there is an error related to the journal paper.

```
public void setJrnPaperErrRC()
```

**setKeyPressedErrorRC():** This method sets a return code for the POSPrinterHandler stating that there is an error as a result of pressing a key while printing.

```
public void setKeyPressedErrorRC()
```

**setDIStationErrorRC():** This method sets a return code for the POSPrinterHandler stating that there is an error on the document insert (DI) station.

```
public void setDIStationErrorRC()
```

**setDIDocMissingRC():** This method sets a return code for the POSPrinterHandler stating that there is no document in the document insert (DI) station.

```
public void setDIDocMissingRC()
```

**setDIDocNotAllowedRC():** This method sets a return code for the POSPrinterHandler stating that the document is not allowed in the document insert (DI) station.

```
public void setDIDocNotAllowedRC()
```

**setFontFileNotDownLoadedRC():** This method sets a return code for the POSPrinterHandler stating that the file was not downloaded correctly.

```
public void setFontFileNotDownLoadedRC()
```

**setDocModeNotValidRC():** This method sets a return code for the POSPrinterHandler stating that the document insert (DI) mode is not valid in the DI station.

```
public void setDocModeNotValidRC()
```

**setOverlayCharsNotValidRC():** This method sets a return code for the POSPrinterHandler stating that the Overlay data for reprint characters was invalid.

```
public void setOverlayCharsNotValidRC()
```

**setCoverOpenButPrintLineOkRC():** This method sets a return code for the POSPrinterHandler stating that the print line was successful even though the cover is open.

```
public void setCoverOpenButPrintLineOkRC()
```

**setJrnBufferOverflowRC():** This method sets a return code for the POSPrinterHandler stating that the journal buffer is full.

```
public void setJrnBufferOverflowRC()
```

**setFlashEepromRC():** This method sets a return code for the POSPrinterHandler stating that there was an error writing to the flash EPROM sector.

```
public void setFlashEepromRC()
```

**setMicrReadErrorRC():** This method sets a return code for the POSPrinterHandler stating that there was an error while flipping the check or performing a MICR read.

```
public void setMicrReadErrorRC()
```

**setNoPowerErrorRC():** This method sets a return code for the POSPrinterHandler stating that the power management feature has removed power from the printer.

```
public void setNoPowerErrorRC()
```

**set4689CRNoPaperRC():** This method sets a return code for the POSPrinterHandler stating that the receipt station on the 4689 printer is out of paper.

```
public void set4689CRNoPaperRC()
```

**set4689SJNoPaperRC():** This method sets a return code for the POSPrinterHandler stating that the summary journal (SJ) station on the 4689 printer is out of paper.

```
public void set4689SJNoPaperRC()
```

**setPrinterOrDriverBufferRC():** This method sets a return code for the POSPrinterHandler stating that the printer buffer is full or the driver buffer is full.

```
public void setPrinterOrDriverBufferRC()
```

**setPrinterNoStorageRC():** This method sets a return code for the POSPrinterHandler stating that the printer station storage is unavailable.

```
public void setPrinterNoStorageRC()
```

**setInvalidOptionRC():** This method sets a return code for the POSPrinterHandler stating that an invalid option was specified for a printer station.

```
public void setInvalidOptionRC()
```

**setNoResourcesRC():** This method sets a return code for the POSPrinterHandler stating that there are not enough system resources to satisfy this request.

```
public void setNoResourcesRC()
```

**setReceivedIllegalParamRC():** This method sets a return code for the POSPrinterHandler stating that an illegal parameter was received.

```
public void setReceivedIllegalParamRC()
```

## Handler Implementation

Following are examples for implementation for the following handlers:

- ANDisplay1/ANDisplay2 Handler
- CashReceipt Monitor Handler
- IOProcessor/ExtendedIOProcessor Handler
- Magnetic Stripe Reader (MSR) Handler
- POSPrinter Handler
- ANDisplay1/ANDisplay2 Handlers using JavaPOS drivers

A demo application called DMTester is provided as a sample application that uses these handlers.

### ANDisplay1/ANDisplay2 Handler Implementation:

```
// *****
//
// ANDImp.java
//
// This is a sample implementation of the ANDisplayHandler interface
// This code allows you to implement the full capability of the
// ANDisplay1 in // Java.
//
// *****

import com.ibm.OS4690.*;

public class ANDImp implements ANDisplayHandler
{
    short m_sRow      = 0;    //current row
    short m_sCol      = 0;    //current column
    byte[] m_byArrData = null; //data buffer to hold display data

    /**
     * The default constructor
     */
    public ANDImp()
    {
        m_byArrData = new byte[40]; //assume it is a 2X20 display
    }

    /**
     * Open the display
     * @exception ANDisplayHandlerException if unable to complete the open
     * request.
     */
    public void open() throws ANDisplayHandlerException
    {
        System.out.println("ANDImp: <open> Method\n");
    }

    /**
     * Close the display
     */
    public void close()
    {
        System.out.println("ANDImp: <close> Method\n");
    }

    /**
     * Clear the display
     * @exception ANDisplayHandlerException if unable to complete the clear
     * request.
     */
}
```

```

public void clear() throws ANDisplayHandlerException
{
    System.out.println("ANDImp: <clear> Method\n");
    for (int i = 0 ; i < m_byArrData.length ; i++) m_byArrData[i] = 0x20;
}

/**
 * Set the cursor to a specific position on the display
 * @param row the current row
 * @param column the current column
 * @exception ANDisplayHandlerException if the position is invalid
 */
public void locate(short row,short column)
throws ANDisplayHandlerException
{
    if ((row < 0 || row > 2) && (column < 0 || column > 40))
    {
        ANDisplayHandlerException ex = new ANDisplayHandlerException();
        ex.setInvalidCursorPositionRC();
        throw ex;
    }

    m_sRow = row;
    m_sCol = column;

    System.out.println("ANDImp: <locate> Method\n");
    System.out.println("\trow    = " + row);
    System.out.println("\tcolumn = " + column);
}

/**
 * Get the current row
 * @return the current row
 */
public short getRow()
{
    System.out.println("ANDImp: <getRow> Method\n");
    System.out.println("\trow = " + m_sRow);

    return m_sRow;
}

/**
 * Get the current column
 * @return the current column
 */
public short getColumn()
{
    System.out.println("ANDImp: <getColumn> Method\n");
    System.out.println("\tcolumn = " + m_sCol);

    return m_sCol;
}

/**
 * Get the current display type
 * @return the current display type
 */
public short getDisplayType()
{
    System.out.println("ANDImp: <getDisplayType> Method\n");
    System.out.println("\tGetDisplayType. displayType = 0x20");

    return 0x20;
}

/**

```

```

*   Read the data displayed on the display
*   @param buf a buffer to hold the display data
*   @return the length of the data read
*   @exception ANDisplayHandlerException if the buffer size is incorrect.
*/
public int read(byte[] buf) throws ANDisplayHandlerException
{
    System.out.println("ANDImp: <read> Method\n");
    if (buf.length > m_byArrData.length)
    {
        ANDisplayHandlerException ex =
            new ANDisplayHandlerException("Invalid buffer size for read");
        ex.setDefaultRC();
        throw ex;
    }

    for (int i = 0 ; i < buf.length ; i++) buf[i] = m_byArrData[i];

    System.out.println("Data read from the display\n");
    System.out.println(new String(buf));

    return buf.length;
}

/**
*   Write the data on the display
*   @param buf a buffer with data to be displayed
*   @return the length of the data written to the display
*   @exception ANDisplayHandlerException if the buffer size is incorrect.
*/
public int write(byte[] buf) throws ANDisplayHandlerException
{
    System.out.println("ANDImp: <write> Method\n");

    if (buf.length > m_byArrData.length)
    {
        ANDisplayHandlerException ex =
            new ANDisplayHandlerException("Invalid buffer size for"+
                                           " write");
        ex.setDefaultRC();
        throw ex;
    }

    for (int i = 0 ; i < buf.length ; i++) m_byArrData[i] = buf[i];

    System.out.println("Data to be written to the display\n");
    System.out.println(new String(buf));

    return buf.length;
}
}

```

#### CashReceipt Monitor Handler Implementation:

```

// *****
//
// CRMImp.java
//
// This is a sample implementation of the CashReceiptMonitor interface.
// This code allows you to monitor the activities on the Cash Receipt station
// of the printer. You are allowed to monitor the data only - the data
// cannot be altered.
//
// *****

import com.ibm.OS4690.*;

```

```

/**
 * This Interface allows the Java GUI to get a copy of data that was
 * sent to the printer.
 */
public class CRMImp implements CashReceiptMonitor
{
    /**
     * The default constructor
     */
    public CRMImp()
    {
        System.out.println("CRMImp: Constructor\n");
    }

    /**
     * The data that should be send to the CashReceipt station
     * @param buf the data buffer that contains the data
     */
    public void printData(byte[] buf)
    {
        System.out.println("CRMImp: <printData> Method\n");
        System.out.println("The data to be printed\n");
        System.out.println(new String(buf));
    }
}

```

#### **IOProcessor/ExtendedIOProcessor Handler Implementation:**

```

// *****
//
// ExIOPImp.java
//
// This is a sample implementation of the new ExtendedIOPHandler interface.
// This code allows you to implement the full capability of the IOP in Java.
//
// *****

import com.ibm.OS4690.*;

//If using the old IOPHandler then implement that handler as shown below
//public class ExIOPImp implements IOPHandler
//else implement the new ExtendedIOPHandler as shown below
public class ExIOPImp implements ExtendedIOPHandler
{
    //is the device locked?
    private boolean m_bLocked          = false;
    //is in priority mode?
    private boolean m_bPriority         = false;
    //is in purge mode?
    private boolean m_bPurged          = false;
    //the current state
    private short   m_sState            = 1;
    //set to TRUE if the IO processor was successfully opened
    private boolean m_bOpened           = false;
    //has the state table loaded successfully?
    private boolean m_bISTableLoaded    = false;
    //has the format table loaded successfully?
    private boolean m_bFTableLoaded     = false;
    //has the modulo table loaded successfully?
    private boolean m_bMTableLoaded     = false;
    //current buffer size
    private int     m_nBufferSize       = 0;
    //total number of buffers
    private short   m_sNumBuffers       = 0;
    //is IOP data available (from keyboard or scanner)

```

```

private boolean m_bDataAvailable = false;
//set to TRUE so that we can remember that the basic application
//is expecting data from us
boolean m_bDataRequested = false;
//a buffer to hold data
private byte[] m_byData = null;

/**
 * The default constructor
 */
public ExIOImp() {}

/**
 * Open the IO processor
 * @exception IOPHandlerException if the IOP is already open or
 * if the state table is not loaded.
 */
public void open() throws IOPHandlerException
{
    System.out.println("IOImp: <open> Method\n");

    if (m_bOpened)
    {
        IOPHandlerException iop = new IOPHandlerException();
        iop.setIOPOpenErrorRC();
        throw iop;
    }

    if (!m_bISTableLoaded)
    {
        IOPHandlerException iop = new IOPHandlerException();
        iop.setInputStateTableNotLoadedRC();
        throw iop;
    }

    m_bOpened = true;
}

//WARNING: This method should be commented out if you are implementing the
//old IOPHandler interface. Although there is no harm in leaving this
//method uncommented, it will never be invoked if this class implements the
//old IOPHandler.
/**
 * Perform a write on the IO processor
 * @param buf the data buffer containing data to be written
 * @exception IOPHandlerException if the write operation could not be
 * completed.
 */
public int write(byte[] buf) throws IOPHandlerException
{
    System.out.println("IOImp: <write> Method\n");
    if (buf.length > m_nBufferSize)
    {
        IOPHandlerException ex = new IOPHandlerException();
        ex.setWriteBufferNotValidRC();
        throw ex;
    }

    m_byData = new byte[buf.length];
    for (int i = 0 ; i < buf.length ; i++) m_byData[i] = buf[i];

    System.out.println("Here is the data to be written\n");
    System.out.println(new String(buf));

    return buf.length;
}

```

```

//WARNING: This method should be commented out if you are implementing the
//old IOPHandler interface. Although there is no harm in leaving this
//method uncommented, it will never be invoked if this class implements the
//old IOPHandler.
/**
 * Obtain the current status of the IOP Handler.
 * @return a 4-byte integer with status information.
 * Note: return -1 if the DeviceManager should construct the
 * correct status information automatically. If not, you must
 * construct it as specified in the programmers guide.
 * @exception IOPHandlerException if unable to complete this request.
 */
public int getStatus() throws IOPHandlerException
{
    System.out.println("IOImp: <getStatus> Method\n");

    /*
    This method returns the current status as a 4-byte integer.
    The format of this integer is FFTTSLSH as described in the CBASIC
    language reference. By returning -1 you can let the DeviceManager
    construct the 4 status bytes. Otherwise, the value returned by this
    method will be used. If the returned value is -1, the DeviceManager
    will construct the 4 status bytes as it was done in the old IOPHandler
    interface. In other words, the following 4 methods will be used as
    appropriate to construct the 4 status bytes:

isDeviceLocked()
isDevicePurged()
isDeviceInPriorityMode()
getLastStateRead()
    */

    return -1;
}

/**
 * Called by DeviceManager to configure current buffers
 * @param numberOfBuffers the total number of buffers
 * @param bufferSize the buffer size
 */
public void configureBuffers(short numberOfBuffers, int bufferSize)
{
    System.out.println("IOImp: <configureBuffers> Method\n");
    System.out.println("number of buffers = " + numberOfBuffers);
    System.out.println("buffSize      = " + bufferSize);

    m_nBufferSize = bufferSize;
    m_sNumBuffers = numberOfBuffers;

    m_byData = new byte[m_nBufferSize];

    for (int i = 0; i < m_nBufferSize; i++) m_byData[i] = 0x00;
}

/**
 * Close the IO processor
 */
public void close()
{
    System.out.println("IOImp: <close> Method\n");
    m_bOpened = false;
}

/**
 * Is the device locked
 * @return true if the device is locked
 */

```



```

public boolean isDeviceLocked()
{
    System.out.println("IOImp: <isDeviceLocked> Method\n");
    System.out.println("locked = " + m_bLocked);
    return m_bLocked;
}

/**
 * Is the device in priority mode
 * @return true if the device is in priority mode
 */
public boolean isDeviceInPriorityMode()
{
    System.out.println("IOImp: <isDeviceInPriorityMode> Method\n");
    System.out.println("priority = " + m_bPriority);
    return m_bPriority;
}

/**
 * Is the device purged
 * @return true if the device is purged
 */
public boolean isDevicePurged()
{
    System.out.println("IOImp: <isDevicePurged> Method\n");
    System.out.println("purged = " + m_bPurged);
    return m_bPurged;
}

/**
 * Return the last state read
 * @return the last state read
 */
public short getLastStateRead()
{
    System.out.println("IOImp: <getLastStateRead> Method\n");
    System.out.println("last state = " + m_sState);
    return m_sState;
}

/**
 * Load the input state table
 * @param tableName the byte array containing the table name
 * @exception IOPHandlerException if the table was not found.
 */
public void loadInputStateTable(byte[] tableName)
    throws IOPHandlerException
{
    System.out.println("IOImp: <loadInputStateTable> Method\n");
    System.out.println("table name = " + new String(tableName));
    System.out.println("table size = " + tableName.length);

    if (!(new java.io.File(new String(tableName)).exists()))
    {
        IOPHandlerException iop = new IOPHandlerException();
        iop.setTableNotFoundRC();
        throw iop;
    }

    m_bISTableLoaded = true;
}

/**
 * Load the format state table
 * @param tableName the byte array containing the table name
 * @exception IOPHandlerException if the table was not found.
 */

```

```

public void loadFormatTable(byte[] tableName) throws IOPHandlerException
{
    System.out.println("IOImp: <loadFormatTable> Method\n");
    System.out.println("table name = " + new String(tableName));
    System.out.println("table size = " + tableName.length);

    if (!(new java.io.File(new String(tableName)).exists()))
    {
        IOPHandlerException iop = new IOPHandlerException();
        iop.setTableNotFoundRC();
        throw iop;
    }

    m_bFTableLoaded = true;
}

/**
 * Load the modulo state table
 * @param tableName the byte array containing the table name
 * @exception IOPHandlerException if the table was not found.
 */
public void loadModuloTable(byte[] tableName) throws IOPHandlerException
{
    System.out.println("IOImp: <loadModuloTable> Method\n");
    System.out.println("table name = " + new String(tableName));
    System.out.println("table size = " + tableName.length);

    if (!(new java.io.File(new String(tableName)).exists()))
    {
        IOPHandlerException iop = new IOPHandlerException();
        iop.setTableNotFoundRC();
        throw iop;
    }

    m_bMTableLoaded = true;
}

/**
 * Lock IOP with purge
 * @param purge if true lock with purge
 */
public void lockDevice(boolean purge)
{
    System.out.println("IOImp: <lockDevice> Method\n");
    System.out.println("purge = " + purge);

    m_bLocked = true;
    m_bPurged = purge;

    if (purge && m_byData != null)
        for (int k = 0; k < m_byData.length; k++) m_byData[k] = 0x20;
}

/**
 * Request a read from the IO processor
 * @param buf a buffer to hold the data read
 * @return the length of the data read
 * @exception IOPHandlerException if the buffer is too big.
 */
public int read(byte[] buf) throws IOPHandlerException
{
    System.out.println("IOImp: <read> Method\n");
    if ((m_byData == null) || (buf.length > m_byData.length))
    {
        IOPHandlerException ex = new IOPHandlerException();
        ex.setBufferOverflowRC();
        throw ex;
    }
}

```

```

    }

    if (m_bLocked)
    {
        IOHandlerException iop = new IOHandlerException();
        iop.setDriverLockedRC();
        throw iop;
    }

    //you should make sure to return the data as specified
    //in the CBASIC language reference - IOP expects upto 10 strings.
    //The first one contains the header data. Also, the data
    //should end with a carriage return+newline (0x0D, 0x0A)
    for (int i = 0; (i < m_byData.length) && (i < buf.length); i++)
        buf[i] = m_byData[i];
    buf[buf.length-1] = 0x0A;
    buf[buf.length-2] = 0x0D; //make sure to terminate the data correctly.

    //now that we have copied the data we will reset the flag
    m_bDataAvailable = false;
    m_bDataRequested = false;
    System.out.println("data read = " + new String(buf));
    System.out.println("bytes read = " + buf.length);
    return buf.length;
}

/**
 * Unlock the IO processor
 * @exception IOHandlerException if unable to unlock the IOP.
 */
public void unlock() throws IOHandlerException
{
    System.out.println("IOImp: <unlock> Method\n");
    m_bLocked = false;
}

/**
 * Unlock the IO processor to a state
 * @param state the state to unlock IOP
 * @exception IOHandlerException if unable to unlock the IOP.
 */
public void unlock(short state) throws IOHandlerException
{
    System.out.println("IOImp: <unlock> Method\n");
    System.out.println("state = " + state);

    if (state != 1)
    {
        IOHandlerException iop = new IOHandlerException();
        iop.setStateNotValidRC();
        throw iop;
    }

    m_bLocked = false;
    m_sState = state;
}

/**
 * Unlock the IO processor to a state using priority
 * @param state the state to unlock IOP
 * @param priority if true unlock with priority
 * @exception IOHandlerException if unable to unlock the IOP.
 */
public void unlock(short state, boolean priority)
throws IOHandlerException
{
    System.out.println("IOImp: <unlock> Method\n");

```

```

        System.out.println("state    = " + state);
        System.out.println("priority = " + priority);

        if (state != 1)
        {
            IOPHandlerException iop = new IOPHandlerException();
            iop.setStateNotValidRC();
            throw iop;
        }

        m_bLocked    = false;
        m_sState     = state;
        m_bPriority   = priority;
    }

    /**
     * Request the IOPHandler to perform an unlock.
     * @param sState The new state you wish to unlock to.
     * @param byTT the byte containing the TAB order
     * @param byFF the byte containing the status information
     * @exception IOPHandlerException if unable to unlock to the specified
     * state.
     */
    public void unlock(short sState, byte byTT, byte byFF)
    throws IOPHandlerException
    {
        System.out.println("IOImp: <unlock> Method\n");
        System.out.println("Requesting the IOP handler to perform an "+
            "unlock to a specifc state and tab order\n");
        System.out.println("\tstate      = "+sState);
        System.out.println("\tttab order = "+byTT);
        System.out.println("\tstatus   = "+byFF);
    }

    /**
     * Return true if IOP data is available to be read
     * @return true if IOP data is available
     */
    public boolean isIOPDataAvailable()
    {
        System.out.println("IOImp: <isIOPDataAvailable> Method\n");
        System.out.println("dataAvailable = " + m_bDataAvailable);
        /*
         This method will be called by the DM when the basic application is
         expecting some data - i.e., from the keyboard, scanner, etc. At this
         time, we should make a note of this request for later - here we will
         simply set the flag to true so that when the data becomes available we
         can notify the basic application.

         If there is already data pending to be read, we should return true here
         so that the basic application can read it immediately. Depending upon the
         basic application, if there is no data this method may get called
         repeatedly until either data is available or a timeout occurs. If the
         timeout occurs, you should prompt the user to enter data or take some
         appropriate action.

         If the data is not available and if the flag is true then you should
         notify the DM as soon as the data becomes available. At this time, you
         should also set the flag to false (a good place for this is the read
         method).
        */
        m_bDataRequested = true;
        return m_bDataAvailable;
    }

    /**
     * Set the keyboard data

```

```

    * @param fileName the keyboard file name
    * @param index the index
    */
    public void setKeyBoardData(String fileName, short index)
    {
        System.out.println("IOImp: <setKeyBoardData> Method\n");
        System.out.println("fileName = " + fileName);
        System.out.println("index    = " + index);
    }
}

```

#### Magnetic Stripe Reader (MSR) Handler Implementation:

```

// *****
//
// MSRImp.java
//
// This is a sample implementation of the MSRHandler interface.
// This code allows you to implement the full capability of the MSR in Java.
//
// *****

import com.ibm.OS4690.*;

public class MSRImp implements MSRHandler
{
    //a data buffer to hold data (size is set to an arbitrary value)
    byte[] m_byData = new byte[255];
    //is MSR data available?
    boolean m_bDataAvailable = false;
    //is MSR Locked?
    boolean m_bDeviceLocked = false;
    //did the basic application request data?
    boolean m_bMSRDataRequested = false;

    /**
     * The default constructor
     */
    public MSRImp(){}

    /**
     * Open the MSR
     * @exception MSRHandlerException if unable to open the MSR.
     */
    public void open() throws MSRHandlerException
    {
        System.out.println("MSRImp: <open> Method\n");
    }

    /**
     * Close the MSR
     * @exception MSRHandlerException if unable to close the MSR.
     */
    public void close() throws MSRHandlerException
    {
        System.out.println("MSRImp: <close> Method\n");
    }

    /**
     * Lock the MSR
     * @param bLock if true lock MSR else unlock it
     * @exception MSRHandlerException if unable to lock the MSR.
     */
    public void lock(boolean bLock) throws MSRHandlerException
    {
        System.out.println("MSRImp: <lock> Method\n");
    }
}

```

```

        m_bDeviceLocked = bLock;
    }

/**
 * Read the data from the MSR
 * @param buf a buffer to hold the MSR data
 * @return the length of the data read
 * @exception MSRHandlerException if the buffer is too big
 */
public int read(byte[] buf) throws MSRHandlerException
{
    System.out.println("MSRImp: <read> Method\n");

    //check the buffer size
    if (buf.length > m_byData.length)
    {
        MSRHandlerException ex = new MSRHandlerException();
        ex.setBufferTooSmallRC();
        throw ex;
    }

    //check if the MSR is locked
    if (m_bDeviceLocked)
    {
        MSRHandlerException ex = new MSRHandlerException();
        ex.setDeviceOfflineRC();
        throw ex;
    }

    //copy the data
    for (int i = 0 ; (i < m_byData.length) && (i < buf.length) ; i++)
        buf[i] = m_byData[i];

    //now that we have copied the data we will reset the flag
    m_bDataAvailable = false;
    m_bMSRDataRequested = false;

    System.out.println("data from the MSR\n");
    System.out.println(new String(buf));

    return buf.length;
}

/**
 * Return true if MSR data is available to be read
 * @return true if MSR data is availbale
 */
public boolean isMSRDataAvailable()
{
    System.out.println("MSRImp: <isMSRDataAvailable> Method\n");
    System.out.println("dataAvailable = " + m_bDataAvailable);
    /*
    This method will be called by the DM when the basic application is
    expecting some data - i.e., from the keyboard, scanner, etc. At this
    time, we should make a note of this request for later - here we will
    simply set the flag to true so that when the data becomes available we
    can notify the basic application.

    If there is already data pending to be read, we should return true here
    so that basic application can read it immediately. Depending upon the
    basic application, if there is no data this method may get called
    repeatedly until either data is available or a timeout occurs. If the
    timeout occurs, you should prompt the user to enter data or take some
    appropriate action.

    If the data is not available and if the flag is true then you should
    notify the DM as soon as the data becomes available. At this time, you

```

```

        should also set the flag to false (a good place for this is the read
        method).
        */

        m_bMSRDataRequested = true;
        return m_bDataAvailable;
    }

    /**
     * Return true if the MSR is locked
     * @return true if MSR is locked else return false
     */
    public boolean isDeviceLocked()
    {
        System.out.println("MSRImp: <isDeviceLocked> Method\n");
        System.out.println("MSR locked = " + m_bDeviceLocked);
        return m_bDeviceLocked;
    }

    /**
     * Return the status information as a long integer
     * @return the status as a 4-byte integer
     * Note: See the BASIC language reference for the format of these bytes
     * @exception MSRHandlerException if unable to complete this request.
     */
    public int getStatus() throws MSRHandlerException
    {
        //The return value is a 4 byte integer of the form RR,SS,RR,LL
        //For example
        byte RR = 0; //reserved byte
        byte SS = 0; //status byte
        byte FF = 0; //format byte
        byte LL = 0; //locked/unlocked?

        LL = (m_bDeviceLocked ? (byte)1 : (byte)0);

        int nRet = (RR << 24)+(SS << 16)+(FF << 8)+LL;

        return nRet;
    }
}

```

#### POSPrinter Handler Implementation:

```

// *****
//
// PRNImp.java
//
//
// This is a sample implementation of the POSPrinterHandler interface.
// This code allows you to implement the full capability of the POSPrinter in
// Java.
//
// *****

import com.ibm.OS4690.*;

public class PRNImp implements POSPrinterHandler
{
    //is data available?
    boolean m_bDataAvailable = false;
    //did basic application request data?
    boolean m_bDataRequested = false;
    //a byte array to hold data (a single array is used here to hold data for
    //all 3 stations)
    byte[][] m_byData = null;
    //station name (can be Cash Receipt, Document Insert, or Summary Journal)

```

```

String m_strStation    = "";

/**
 * The default constructor
 */
public PRNimp(){}

/**
 * Open specified station on the printer
 * @param nStation printer station - it can be cash receipt,
 * document insert, or summary journal station
 * @exception POSPrinterHandlerException if unable to open the specified
 * station.
 */
public void open(int nStation) throws POSPrinterHandlerException
{
    System.out.println("PRNimp: <open> Method\n");
    stationIDtoString(nStation);
    System.out.println("station = "+m_strStation);

    //initialize the data buffer
    if (nStation == DeviceManagerConst.PS_CASH_RECEIPT)
        m_byData[0] = new byte[DeviceManagerConst.CR_MAX_BUF_LENGTH];
    else if (nStation == DeviceManagerConst.PS_DOCUMENT_INSERT)
        m_byData[1] = new byte[DeviceManagerConst.DI_MAX_BUF_LENGTH];
    else
        m_byData[2] = new byte[DeviceManagerConst.SJ_MAX_BUF_LENGTH];
}

/**
 * Close the specified station
 * @param nStation printer station - it can be cash receipt,
 * document insert, or summary journal station
 * @param bTClose if true perform a TCLOSE else do a CLOSE
 * @exception POSPrinterHandlerException if unable to close the
 * specified station.
 */
public void close(int nStation, boolean bTClose)
throws POSPrinterHandlerException
{
    System.out.println("PRNimp: <close> Method\n");
    System.out.println("station = "+m_strStation);
}

/**
 * Request the PrinterHandler perform the write.
 * @param buf buffer containing the data to write
 * @return number of bytes written
 * @exception PrinterHandlerException if unable to complete the write
 * operation.
 */
public int write(int nStation, byte[] buf)
throws POSPrinterHandlerException
{
    System.out.println("PRNimp: <write> Method\n");
    System.out.println("station = "+m_strStation);

    int nIndex = 0;
    if (nStation == DeviceManagerConst.PS_CASH_RECEIPT)
nIndex = 0;
    else if (nStation == DeviceManagerConst.PS_DOCUMENT_INSERT)
nIndex = 1;
    else
        nIndex = 2;

    //check the buffer size
    if (buf.length > m_byData[nIndex].length)

```



```

    {
        POSPrinterHandlerException ex = new POSPrinterHandlerException();
        ex.setBufferOverflowRC();
        throw ex;
    }

    //copy the data
    for (int i = 0 ; i < buf.length ; i++) m_byData[nIndex][i] = buf[i];

    //now that we have copied the data we will reset the flag
    m_bDataAvailable = false;
    m_bDataRequested = false;

    System.out.println("data read  = "+new String(buf));
    return buf.length;
}

/**
 * Request the PrinterHandler perform a read.
 * @param buf buffer containing the data to write
 * @return number of bytes read
 * @exception PrinterHandlerException if unable to complete the read
 * operation.
 */
public int read(int nStation, byte[] buf)
throws POSPrinterHandlerException
{
    System.out.println("PRNImp: <read> Method\n");
    System.out.println("station = "+m_strStation);

    int nIndex = 0;
    if (nStation == DeviceManagerConst.PS_CASH_RECEIPT)
nIndex = 0;
    else if (nStation == DeviceManagerConst.PS_DOCUMENT_INSERT)
nIndex = 1;
    else
        nIndex = 2;

    //check the buffer size
    if (buf.length > m_byData[nIndex].length)
    {
        POSPrinterHandlerException ex = new POSPrinterHandlerException();
        ex.setBufferOverflowRC();
        throw ex;
    }

    //copy the data
    for (int i = 0 ; (i < m_byData.length) && (i < buf.length) ; i++)
        buf[i] = m_byData[nIndex][i];

    //now that we have copied the data we will reset the flag
    m_bDataAvailable = false;
    m_bDataRequested = false;

    System.out.println("data read  = "+new String(buf));

    return buf.length;
}

/**
 * Get the current status of the printer
 * @return status bytes as a single four byte integer
 * Note: See the BASIC language reference for the format of these bytes
 * @exception PrinterHandlerException if unable to complete this request.
 */
public int getStatus(int nStation) throws POSPrinterHandlerException
{

```

```

        System.out.println("PRNImp: <getStatus> Method\n");
        System.out.println("station = "+m_strStation);
        System.out.println("status = 0");
        return 0;
    }

    /**
     * Set the printer status
     * @param nStatus a four byte integer with status bytes
     * Note: See the BASIC language reference for the format of these bytes.
     * @exception PrinterHandlerException if unable to complete this
     * request..
     */
    public void setStatus(int nStation, int nStatus)
    throws POSPrinterHandlerException
    {
        System.out.println("PRNImp: <setStatus> Method\n");
        System.out.println("station = "+m_strStation);
        System.out.println("status = "+nStatus);
    }

    /**
     * Find out if data is available or there is more data
     * @return true if data is available else return false
     */
    public boolean isPOSPrinterDataAvailable()
    {
        System.out.println("PRNImp: <isMICRDataAvailable> Method\n");
        System.out.println("dataAvailable = "+m_bDataAvailable);
        /*
         This method will be called by the DM when the basic application is
         expecting some data - i.e., from the keyboard, scanner, etc. At this
         time, we should make a note of this request for later - here we will
         simply set the flag to true so that when the data becomes available we
         can notify the basic application.

         If there is already data pending to be read we should return true here
         so that basic application can read it immediately. Depending upon the
         basic application, if there is no data this method may get called
         repeatedly until either data is available or a timeout occurs. If the
         timeout occurs you should prompt the user to enter data or take some
         appropriate action.

         If the data is not available and if the flag is true then you should
         notify the DM as soon as the data becomes available. At this time you
         should also set the flag to false (a good place for this is the read
         method).
         */
        m_bDataRequested = true;
        return m_bDataAvailable;
    }

    /**
     * Convert station ID to a string (for convenience)
     * @param nStation the printer station
     */
    void stationIDtoString(int nStation)
    {
        m_byData = new byte[3][0];
        if (nStation == DeviceManagerConst.PS_CASH_RECEIPT)
        {
            m_byData[0] = new byte[DeviceManagerConst.CR_MAX_BUF_LENGTH];
            m_strStation = "Cash Receipt Station";
        }
        else if (nStation == DeviceManagerConst.PS_DOCUMENT_INSERT)
        {
            m_byData[1] = new byte[DeviceManagerConst.DI_MAX_BUF_LENGTH];

```

```

        m_strStation = "Document Insert Station";
    }
    else if (nStation == DeviceManagerConst.PS_SUMMARY_JOURNAL)
    {
        m_byData[2] = new byte[DeviceManagerConst.SJ_MAX_BUF_LENGTH];
        m_strStation = "Summary Journal Station";
    }
}
}

```

#### ANDDisplay1/ANDDisplay2 Handler Implementation that makes use of JavaPOS drivers:

```

// *****
//
// ANDJposImp.java
//
//
// This is a sample implementation of the ANDDisplayHandler interface.
// This code allows you to implement the full capability of the ANDDisplay1
// or the ANDDisplay2 in Java.
// Here we use Toshiba JavaPOS drivers to test the implementation by redirecting
// the redirected i/o to the actual display
//
// *****

import com.ibm.OS4690.*;
import jpos.*;
import jpos.events.*;
import jpos.services.*;

public class ANDJposImp implements ANDDisplayHandler
{
    short m_sRow    = 0;    //current row
    short m_sCol    = 0;    //current column
    byte[] m_byArrData = null; //data buffer to hold display data

    LineDisplay m_jpDisplay = null; //the JavaPOS ANDDisplay object

    /**
     * The default constructor
     */
    public ANDJposImp()
    {
        m_byArrData = new byte[40]; //assume it is a 2X20 display
    }

    /**
     * Open the display
     * @exception ANDDisplayHandlerException if unable to open the display.
     */
    public void open() throws ANDDisplayHandlerException
    {
        System.out.println("ANDJposImp: <open> Method\n");
        try
        {
            //logical name depends on the display type
            //it can be either ADXPiA0: or ADXPiE0:
            m_jpDisplay.open("ADXPiA0:"); // open the jpos line display
            m_jpDisplay.claim(1000); // claim the device
            m_jpDisplay.setDeviceEnabled(true); // enable the device
            m_sRow = 1;
            m_sCol = 1;
            System.out.println("ANDJposImp: ANDDisplay was opened"+
                               " successfully.\n");
        }
        catch (Exception e)
        {

```

```

        ANDDisplayHandlerException ex =
            new ANDDisplayHandlerException("ANDJposImp: display "+
"cannot be opened: "+e.getMessage());
        ex.setDeviceOfflineRC();
        throw ex;
    }
}

/**
 * Close the display
 */
public void close()
{
    System.out.println("ANDJposImp: <close>Method\n");
    try
    {
        m_jpDisplay.release();        // release the jpos line display
        m_jpDisplay.close();          // close the device
        System.out.println("ANDJposImp: ANDDisplay was closed"+
" successfully.\n");
    }
    catch (Exception e)
    {
        System.out.println("ANDJposImp: display cannot be closed: "+
e.getMessage());
    }
}

/**
 * Clear the display
 * @exception ANDDisplayHandlerException if unable to complete this
 * request.
 */
public void clear() throws ANDDisplayHandlerException
{
    System.out.println("ANDJposImp: <clear> Method\n");
    for (int i = 0 ; i < m_byArrData.length ; i++)
m_byArrData[i] = 0x20;

    try
    {
        m_jpDisplay.clearText(); // clear all the text on the Line
        // Display
    }
    catch (Exception e)
    {
        ANDDisplayHandlerException ex =
            new ANDDisplayHandlerException("ANDJposImp: clear method"+
" failed: "+e.getMessage());
        ex.setDefaultRC();
        throw ex;
    }
}

/**
 * Set the cursor to a specific position on the display
 * @param row the current row
 * @param column the current column
 * @exception ANDDisplayHandlerException if the position (row or column)
 * is invalid.
 */
public void locate(short row,short column)
throws ANDDisplayHandlerException
{
    if ((row < 0 || row > 2) && (column < 0 || column > 40))
    {
        ANDDisplayHandlerException ex = new ANDDisplayHandlerException();
    }
}

```

```

        ex.setInvalidCursorPositionRC();
        throw ex;
    }

    m_sRow = row;
    m_sCol = column;

    System.out.println("ANDJposImp: <locate> Method\n");
    System.out.println("\trow    = " + row);
    System.out.println("\tcolumn = " + column);

    try
    {
        m_jpDisplay.setCursorRow(m_sRow);    // set the current row
        m_jpDisplay.setCursorColumn(m_sCol);  // set the current column
    }
    catch (Exception e)
    {
        ANDDisplayHandlerException ex =
            new ANDDisplayHandlerException("ANDJposImp: locate method"+
" failed: "+e.getMessage());
        ex.setDefaultRC();
        throw ex;
    }
}

/**
 * Get the current row
 * @return the current row
 */
public short getRow()
{
    System.out.println("ANDJposImp: <getRow> Method\n");
    System.out.println("\trow = " + m_sRow);

    short sRow = -1;
    try
    {
        // get the current row
        sRow = (short) m_jpDisplay.setCursorRow();
    }
    catch(JposException e)
    {
        e.printStackTrace();
        System.out.println("caught a JposException in getRow():"+
            e.getMessage());
    }

    return sRow;
}

/**
 * Get the current column
 * @return the current column
 */
public short getColumn()
{
    System.out.println("ANDJposImp: <getColumn> Method\n");
    System.out.println("\tcolumn = " + m_sCol);

    short sCol = -1;

    try
    {
        // get the current column
        sCol = (short) m_jpDisplay.setCursorColumn();
    }

```

```

        catch (Exception e)
        {
            e.printStackTrace();
            System.out.println("caught a JposException in getColumn():"+
                               e.getMessage());
        }

        return sCol;
    }

    /**
     * Get the current display type
     * @return the current display type
     */
    public short getDisplayType()
    {
        System.out.println("ANDJposImp: <getDisplayType> Method\n");
        System.out.println("displayType = 0x20");

        return 0x20;
    }

    /**
     * Read the data displayed on the display
     * @param buf a buffer to hold the display data
     * @return the length of the data read
     * @exception ANDisplayHandlerException if the buffer size is incorrect.
     */
    public int read(byte[] buf) throws ANDisplayHandlerException
    {
        System.out.println("ANDJposImp: <read> Method\n");
        if (buf.length > m_byArrData.length)
        {
            ANDisplayHandlerException ex =
                new ANDisplayHandlerException("Invalid buffer size for read");
            ex.setDefaultRC();
            throw ex;
        }

        //copy the data from the buffer to the byte array passed to us
        for (int i = 0 ; i < buf.length ; i++) buf[i] = m_byArrData[i];

        System.out.println("Data read from the display\n");
        System.out.println(new String(buf));

        return buf.length;
    }

    /**
     * Write the data on the display
     * @param buf a buffer with data to be displayed
     * @return the length of the data written to the display
     * @exception ANDisplayHandlerException if the buffer size is incorrect.
     */
    public int write(byte[] buf) throws ANDisplayHandlerException
    {
        System.out.println("ANDJposImp: <write> Method\n");

        if (buf.length > m_byArrData.length)
        {
            ANDisplayHandlerException ex =
                new ANDisplayHandlerException("Invalid buffer size for"+
" write");
            ex.setDefaultRC();
            throw ex;
        }
    }

```

```

        //save the data in a buffer
        for (int i = 0 ; i < buf.length ; i++) m_byArrData[i] = buf[i];

        //now send the data to the display
        try
        {
            m_jpDisplay.displayText(
new String(m_byArrData),
LineDisplayConst.DISP_DT_NORMAL);
        }
        catch (Exception e)
        {
            ANDisplayHandlerException ex =
                new ANDisplayHandlerException("ANDJposImp: write method"+
" failed: "+e.getMessage());
            ex.setDefaultRC();
            throw ex;
        }

        System.out.println("Data to be written to the display\n");
        System.out.println(new String(buf));

        return buf.length;
    }
}

```

#### DMTester - a sample application that uses above handlers:

```

// *****
//
// DMTester.java
//
// This is a sample application that provides implementations for all
// Handler interfaces and the Cash Receipt monitor. It does not provide a
// graphical user interface - all output goes to the standard out and error
// streams
//
// *****

import com.ibm.OS4690.*;

public class DMTester
{
    private MSRImp      m_msrImp    = null; //MSR Handler Implementation
    private PRNImp      m_prnImp    = null; //Printer Handler Implementation
    private ExIOImp     m_iopImp     = null; //Extended IOP Handler Imp.
    //private IOImp      m_iopOldImp = null; //Old IOP Handler Impl.
    private CRMImp      m_crmImp     = null; //Cash Receipt Monitor Imp.
    private ANDImp      m_and1Imp    = null; //ANDisplay1 Handler Imp.
    private ANDImp      m_and2Imp    = null; //ANDisplay2 Handler Imp.

    /**
     * The default constructor
     */
    public DMTester()
    {
        System.out.println("=====\n");
        System.out.println("Java Redirection Tester\n");
        System.out.println("=====\n");

        // construct the Handlers
        m_msrImp = new MSRImp(); //MSR Handler Implementation
        //note that only a single IOHandler can be registered - if you try
        //to register both the old and the new handlers only the first attempt
        //will be successful (second attempt will fail).
        m_iopImp = new ExIOImp(); //The new Extended IOP Handler Imp.
    }
}

```

```

//m_iopOldImp = new IOPImp();    //The Old IOP Handler Implementation
m_crmImp      = new CRMImp();    //Cash Receipt Monitor Implementation
m_and1Imp     = new ANDImp();    //ANDisplay1 Handler Implementation
m_and2Imp     = new ANDImp();    //ANDisplay2 Handler Implementation
m_prnImp      = new PRNImp();    //The Printer Handler Implementation

// register the new handlers
//register the MSRImp as a handler for the MSR
DeviceManager.setMSRHandler(m_msrImp);
//register the new ExIOPImp as a handler for the IOP
DeviceManager.setExtendedIOHandler(m_iopImp);
//register the old IOPImp as a handler for the IOP
//DeviceManager.setIOHandler(m_iopOldImp);
//register the PRNImp as a handler for the Printer
DeviceManager.setPOSPrinterHandler(m_prnImp);
//Note: The CashReceipt Station can be either a monitor or a handler -
//Although you can register it as both a monitor and a handler the
//role it will play depends on the configuration (i.e., handler or a
//a monitor but not as a handler and a monitor at the same time.
//register the CRMImp as a monitor for the CashReceipt Station
DeviceManager.setCashReceiptMonitor(m_crmImp);
//register the ANDImp as a handler for the ANDisplay1
DeviceManager.setANDisplay1Handler(m_and1Imp);
//register the ANDImp as a handler for the ANDisplay2
DeviceManager.setANDisplay2Handler(m_and2Imp);

//notify the device manager that the registration process is complete
DeviceManager.setDeviceRegistrationComplete();
}

public static void main (String[] argv)
{
    DMTester dmTest = new DMTester();
}
}

```

---

## Java I/O Processor Functions

The Java I/O processor (JIOP) provides the following general capabilities to a Java user interface:

- Sending data and function codes to POS application
- Notification of state changes in the Java I/O processor
- Notification of keyboard input
- Notification of scanner input
- Notification of I/O processor lock and unlock
- Exposure of allowed function codes
- Exposure of allowed device input
- Notification of I/O processor operator prompt output
- Notification of I/O processor system busy
- Notification of input sequence refresh
- Notification of application initialization completion

The JIOP supports the loading and processing of the enhanced, full-screen state table with some differences. Specifically, the JIOP does not control the positioning of message areas or other GUI components on the screen and it does not write data to the screen. With the new Java capabilities, the Java GUI application is responsible for controlling the layout of GUI components and writing the data to the screen. The JIOP provides accessor methods for all of the new full-screen attributes, such as the message area parameters, so that the Java GUI can use them to control how the screen should be laid out using a Java Layout Manager. In order to make use of this full-screen support, the enhanced, full-screen support option must be set to Y in the Common Information of the Input State Table Utility.



When in a full-screen state, the JIOP keeps track of the current field and maintains the content of each input field as well as the cursor position within each field for the current state. The Java GUI should add itself as a `FullScreenFieldListener` so that the JIOP notifies it of the different field events. The Java GUI should use these events to keep the corresponding GUI input component in sync with the current input field within the JIOP.

The Java I/O processor replaces the I/O processor originally included in the 4690 Operating System for use with a Java GUI. The Java I/O processor does not provide the following functionality, which was provided by the original I/O processor:

- The JIOP supports the ANPOS keyboard attached to port 5 for alphanumeric data entry using the 2x20 sequence table.
- The JIOP has no support for the KEY/SCAN/WAND ahead functionality (entering data with the I/O processor locked).
- The JIOP does not support an OCR reader or a magnetic wand.

---

## Definitions and Concepts

The definitions and concepts for the original I/O processor still apply. The following terms are related to the Java platform:

### Java event

A notification that a property change has occurred in the event source. The JIOP raises various events to inform the Java GUI of important occurrences.

### Java listener

A class that observes events from an event source. The Java GUI must implement listener methods for the JIOP events it is interested in. It must also register to receive specific events.

---

## Functions that the Java GUI Performs

The Java GUI performs these functions:

- JIOP construction
- Sending input to JIOP
- Receiving events from JIOP

## JIOP Construction

The JIOP is a Java class and must be explicitly constructed by the Java User Interface. Once constructed, the JIOP automatically registers with the Device Manager and initialize in response to application invocation. The terminal must be configured with I/O processor redirection turned on for proper operation of the JIOP.

## Sending Input to JIOP

All input is routed through a queue in the JIOP. This queue provides separate methods for the input of data and function codes by the Java GUI.

See “Class `IOInputQueue`” on page 611 for information about the class that performs this function.

## Receiving Events from JIOP

During operation, the JIOP raises events to inform a GUI of important occurrences. To monitor these events, the standard Java mechanism of implementing an interface and registering for an event is used.

## Field Activated Events

Applications that load an enhanced, full-screen state table might utilize states that contain multiple input fields on a screen. The JIOP maintains which input field receives keyboard input. The operator might

change the current field by tabbing forward and backward through the fields. When a field is activated, the field is considered ready to receive keyboard input, and the JIOP raises an event. The GUI might use this event to set cursor focus to a corresponding GUI input component and enable the component for input.

See “Interface FullScreenFieldListener” on page 621 for information about the listener interface for receiving field activation events.

### **Field Deactivated Events**

Applications that load an enhanced, full-screen state table might utilize states that contain multiple input fields on a screen. The JIOP maintains which input field receives keyboard input. The operator might change the current field by tabbing forward and backward through the fields. When a field is activated, the JIOP raises a `FieldDeactivatedEvent` for the previously active field. The GUI might use this event to disable the corresponding GUI input component.

See “Interface FullScreenFieldListener” on page 621 for information about the listener interface for receiving field activation events.

### **Field Data Updated Events**

The JIOP maintains both the content of the input field as well as the cursor position within the field. Keyboard or scanner input might cause data to be added to the field. Certain keys, such as delete, backspace, and clear, also modify both the content of the field and the cursor position within the field. Other keys, such as the right and left arrow keys, modify the cursor position within the field. When the JIOP processes input that modifies the content or cursor position of an input field, the JIOP raises a `FieldDataUpdatedEvent`. The GUI might use this event to modify the contents of the corresponding GUI component, and set the caret position within the GUI component.

When the application issues a `PUTLONG` request following a `WRITE` request to the JIOP, the JIOP first clears the current input sequence and then issues an unlock for the specific state and tab order. The JIOP fires `FieldDataUpdatedEvents` with a source of `IOPInputQueue.CLEARSEQUENCE` when it clears the current input sequence. When the JIOP issues an unlock, the JIOP fires the `StateChangeEvent` followed by one or more `FieldDataUpdatedEvents` to reflect the data updates caused by the `WRITE` request.

Bit 5 of the `PUTLONG` request controls whether the 4690 I/O Processor initializes the screen data and attributes. If bit 5 is not set, the data in the active input sequence buffer is written to the display screen. If bit 5 is set to 1, then the active input sequence buffer is not written to the display. Because the JIOP does not do any direct writes to the screen, this bit value is passed on to the Java GUI in the `FieldDataUpdatedEvent`. The Java GUI can invoke the `getDoNotInitializeScreenData()` that is defined in the `FieldDataUpdatedEvent`. If the method returns true, then the Java GUI does not display the data that is received in the `FieldDataUpdatedEvent`.

See “Interface FullScreenFieldListener” on page 621 for more information about the listener interface for receiving field data updated events for full-screen fields.

See “Interface InputFieldListener” on page 621 for more information about the listener interface for receiving field data updated events for non-full-screen fields.

### **Initialization Events**

The JIOP is not immediately ready for processing after construction. After registering with the Device Manager a series of initialization processes occur between the JIOP and the POS application. These processes include loading the various input tables and loading the keyboard layout. As a result, a Java GUI must wait for the event indicating that the JIOP has completed initialization prior to invoking any JIOP method.

See “Class IOPReadyListener” on page 602 for information about the listener interface for IOP ready events.

## Keyboard Events

The JIOP provides the capability for monitoring and filtering input from the keyboard. When input is received from the keyboard, it is first presented to the Java GUI for display and validation. The Java GUI has the option of accepting or rejecting the input. If rejection is chosen the GUI should notify the user in an appropriate manner.

See “Interface POSKeyListener” on page 622 for more information about the interface for listening for data and function code input from the POS keyboard.

## Prompt Change Events

When the JIOP needs to display an operator prompt based on the Input Tables, it raises an event. The event allows flexibility in the mechanism used to display the prompt. A GUI can use the event to display the prompt in an appropriate GUI control.

See “Class PromptChangeListener” on page 605 for more information about the listener interface for receiving prompt change events.

## Scanner Events

The JIOP provides the capability for monitoring label input from keyboard and the scanner. Notification is also provided when the operator has entered a keyed label which is invalid.

See “Interface LabelInputListener” on page 624 for more information about the interface for listening to bar label input from the scanner and keyboard.

## Shutdown Event

When the JIOP is closed by the POS application, an event is raised by the JIOP. The Java GUI should respond by updating the GUI to reflect that the POS application is no longer running. Note that the POS application can be subsequently restarted, which results in JIOP initialization and initialization events being sent to the Java GUI. If restart occurs, the Java GUI must reestablish its event listeners with the JIOP.

See “Class IOPShutdownListener” on page 608 for more information about the interface for listening for the shutdown (close) of the Java I/O processor.

## State Change Events

The JIOP is a state-transition system. For each state, there is an allowed set of inputs and operations. The JIOP provides the capability for determining the current state of the JIOP and detecting a transition to a new state.

See “Interface StateChangeListener” on page 633 for more information about the listener interface for receiving state change events.

See “Class StateTableProcessor” on page 629 for more information about the state transition engine for the I/O processor.

## AWT Thread Considerations for Event Listeners

For Java GUI applications that use AWT or Swing, it is important to remember that AWT objects are not thread safe. Objects that register as JIOP event listeners are notified of events on a separate JIOP thread. If the Java GUI updates AWT or Swing components on the JIOP thread, concurrency problems are likely. To avoid these concurrency problems, the Java GUI should make use of the `SwingUtilities.invokeLater()` method when handling JIOP events. The use of the `SwingUtilities.invokeLater()` method allows the thread context to be switched to the AWT thread. For more information on the `SwingUtilities` class, see the Swing API specification.

## System Busy Events

An event is raised when the JIOP enters System Busy state. System Busy state can occur when the operator attempts input when the JIOP is locked (input disallowed). The Java User Interface should display an appropriate GUI indicator in response to System Busy.

See “Class SystemBusyListener” on page 609 for more information about the interface for listening for System Busy to occur in the JIOP.

---

## JIOP Configuration

Redirection of the IOPProcessor must be enabled for proper operation of the JIOP.

The Java I/O processor interacts with POS devices using JavaPOS. JavaPOS must be installed and configured to enable JIOP operation. In addition, the following device logical names must be configured within JavaPOS:

Device	Logical name
Keyboard	ADXPIK0:
Scanner	ADXPIS0:
Second Scanner	ADXPIS1:
Keylock (manager key)	IBMKeylock
Tone	IBMToneIndicator

---

## JIOP code example

```
package demo;

import com.ibm.OS4690.jiop.*;
import com.ibm.OS4690.jiop.util.*;
import com.ibm.OS4690.DeviceManager;

/**
 * Simple demonstration of using JIOP, after execution, JIOP will be
 * running on the system as the I/O processor
 *
 * REQUIRED: POS Application
 *          JavaPOS installed/configured
 *          Redirection configured for JIOP
 */

public class Demo implements Runnable,
                           IOPReadyListener,
                           PromptChangeListener
{
    /**
     * Main
     */
    public static void main(String[] args)
    {
        System.out.println("JIOP demo starting");

        // SystemMonitor will enable trace system for uncaught exceptions
        SystemMonitor sm = new SystemMonitor();
        Thread mainThread = new Thread(sm, new Demo());
        mainThread.start();
    }

    public void run()
    {
        // Construct the JIOP and listen for Ready
        IOPReadyListener[] iopRListener = new IOPReadyListener[1];
        iopRListener[0] = this;
        iop = new JIOPProcessor(iopRListener);

        // Inform DM we are ready
        DeviceManager.setDeviceRegistrationComplete();
    }

    /**
     * Indicates the JIOP is ready
     */
    public void iopReady(IOPReadyEvent iopr)
    {
        System.out.println("JIOP ready");
        iop.addPromptChangeListener(this);
    }

    /**
     * Provides information about the loading process
     */
    public void iopStatus(IOPInitStatus iops)
    {
        System.out.println(iops.getStatus());
    }
}
```

```

/**
 * Dump prompt changes to standard out
 *
 */
public void promptChanged(PromptChangeEvent pce)
{
    System.out.println(pce.getPrompt());
}

JIOProcessor iop;
}

```

---

## Java I/O processor packages

Java I/O processor classes are in two packages: `com.ibm.OS4690.jiop` and `com.ibm.OS4690.jiop.util`.

Java I/O processor classes in package `com.ibm.OS4690.jiop` include:

- JIOProcessor
- IOPReadyListener
- IOPReadyEvent
- IOPInitStatus
- PromptChangeListener
- PromptChangeEvent
- IOPShutdownListener
- SystemBusyListener
- SystemBusyEvent
- IOPInputQueue
- QueueStatusChangeListener
- QueueStatusChangeEvent
- POSKeyListener (deprecated)
- POSKeyListenerEx
- LabelInputListener
- LabelInputEvent
- QueueLockedException
- StateForbidsInputException
- StateTableProcessor
- StateChangeListener
- StateChangeEvent
- State
- FunctionCode
- Globals
- InputSequenceFormatter
- InputSequenceClearedListener
- InputSequenceClearedEvent
- FieldActivatedEvent
- FieldDataUpdatedEvent
- FieldDeactivatedEvent
- FullScreenAttributes  
VideoParms
- FullScreenFieldListener

- `FunctionCodeList`
- `InputFieldListener`

Java I/O processor classes in package `com.ibm.OS4690.jiop.util` include:

- `Assert`
- `Debug`
- `Queue`
- `SystemMonitor`
- `Trace`
- `ExceptionListener`
- `ExceptionEvent`

---

## Classes in package `com.ibm.OS4690.jiop`

This section describes Java I/O processor classes in package `com.ibm.OS4690.jiop`.

---

### Class `JIOProcessor`

This class is the Java version of the 4690 I/O processor. It supports a Java GUI controlling the POS application. It collects I/O from the GUI and any attached POS devices for the POS application. After the `IOPReady` event is raised, the I/O processor is ready for input. Prior to that event, the I/O processor is not in a valid state.

### `JIOProcessor` Constructor

There is only one constructor:

#### **`JIOProcessor(IOPReadyListener[])`**

Constructs the Java IO processor. This should be done only once in an application.

#### **`JIOProcessor(IOPReadyListener ior[])`**

Constructs the Java IO processor, which should be done only once in an application. After construction, the IOP is in an invalid state until connected to the POS application.

### `JIOProcessor` Methods

`JIOProcessor` uses the following methods:

#### **`addIOPReadyListener(IOPReadyListener)`**

Add an IOP ready listener

#### **`addIOPShutdownListener(IOPShutdownListener)`**

Add a shutdown listener

#### **`addPromptChangeListener(PromptChangeListener)`**

Add a prompt change listener

#### **`addSystemBusyListener(SystemBusyListener)`**

Add a system busy listener

#### **`getInputQueue()`**

Query the input queue for this IO processor

#### **`getInputSequenceFormatter()`**

Query the input sequence formatter

#### **`getIOPProcessor()`**

Get the currently running `IOPProcessor`

**getPrompt()**

Get the current prompt to display

**getProperties()**

Get the JIOP properties. The properties class contains information regarding the initialization of the JIOP.

**getTableProcessor()**

Query the table processor

**isReady()**

Query if the JIOP is ready

**isSystemBusy()**

Query if the JIOP is in System Busy state

**removeIOPReadyListener(IOPReadyListener)**

Removes a ready listener

**removeIOPShutdownListener(IOPShutdownListener)**

Removes a shutdown listener

**removePromptChangeListener(PromptChangeListener)**

Removes a prompt change listener

**removeSystemBusyListener(SystemBusyListener)**

Removes a system busy listener

**addIOPReadyListener(IOPReadyListener)**

This method adds an iop ready listener.

```
public synchronized void addIOPReadyListener(IOPReadyListener ior)
```

**Parameters:**

**ior** Ready listener

**addIOPShutdownListener (IOPShutdownListener)**

This method adds a shutdown listener.

```
public synchronized void addIOPShutdownListener(IOPShutdownListener sd)
```

**Parameters:**

**sd** Shut down listener

**addPromptChangeListener (PromptChangeListener)**

Adds a prompt change listener

```
public synchronized void addPromptChangeListener(PromptChangeListener pcl)
```

**Parameters:**

**pcl** Prompt change listener

**addSystemBusyListener(SystemBusyListener)**

Adds a system busy listener

```
public synchronized void addSystemBusyListener(SystemBusyListener sbl)
```

**Parameters:**

**sbl** System busy listener



### **getInputQueue()**

Query the Input Queue for this IO Processor.

```
public static IOPInputQueue getInputQueue()
```

**Returns:** This method returns the input queue.

### **getInputSequenceFormatter()**

Query the input sequence formatter

```
public static InputSequenceFormatter getInputSequenceFormatter()
```

**Returns:** This method returns the input sequence formatter.

### **getIOProcessor()**

This method gets the currently running IOProcessor.

```
public static JIOProcessor getIOProcessor()
```

**Returns:** This method returns the I/O processor.

### **getPrompt()**

This method gets the current prompt to display.

```
public String getPrompt()
```

**Returns:** This method returns the current prompt.

### **getProperties()**

This method gets the JIOP properties, the properties class contains information regarding the initialization of the JIOP. The file name of the properties file is expected to be in Java System properties with keyword props.

```
public static Properties getProperties()
```

**Returns:** This method returns JIOP properties.

### **getTableProcessor()**

This method queries the table processor.

```
public static StateTableProcessor getTableProcessor()
```

**Returns:** This method returns the table processor.

### **isReady()**

This method queries if the JIOP is ready, which means, "Is the open completed?"

```
public boolean isReady()
```

**Returns:** This method returns **true** if ready, and **false** otherwise.

### **isSystemBusy()**

This method queries if the JIOP is in System Busy state.

```
public boolean isSystemBusy()
```

**Returns:** This method returns **true** if JIOP is busy.

### **removeIOPReadyListener(IOPReadyListener)**

This method removes a ready listener.

```
public synchronized void removeIOPReadyListener(IOPReadyListener ior)
```

***Parameters:***

**ior**      Ready listener

**removeIOPShutdownListener (IOPShutdownListener)**

This method removes a shutdown listener.

```
public synchronized void removeIOPShutdownListener(IOPShutdownListener sd)
```

***Parameters:***

**sd**      Shut down listener

**removePromptChangeListener (PromptChangeListener)**

This method removes a prompt change listener.

```
public synchronized void removePromptChangeListener(PromptChangeListener pcl)
```

***Parameters:***

**pcl**      Prompt change listener

**removeSystemBusyListener (SystemBusyListener)**

This method removes a system busy listener.

```
public synchronized void removeSystemBusyListener(SystemBusyListener sb1)
```

***Parameters:***

**sb1**      System busy listener

---

## Class FieldActivatedEvent

Instances of this class are used by the Java IO Processor when processing a full-screen state. The active field is the InputField in the full-screen state which can receive keyboard or scanner input. The JIOP raises FieldActivatedEvents to notify the GUI which field is ready to receive input. The GUI can respond by enabling and setting focus on a corresponding GUI component.

FullScreenFieldListeners can receive FieldActivatedEvents.

The JIOP raises a FieldActivatedEvent in the following situations:

- The JIOP receives a request to unlock into a full-screen state. If a tab order is indicated on the unlock request, a FieldActivatedEvent is generated for the InputField of the function code with the requested tab order in the state. If no function code in the state is defined with the requested tab order, a FieldActivatedEvent is raised for the InputField of the function code with the *lowest* tab order.
- The operator presses a forward or backward tab key to navigate from one field to another field.
- A full-screen function code is defined with autotabbing enabled, and the operator reaches the maximum data length for the field. A FieldActivatedEvent is generated for the next field in the tab order.

## FieldActivatedEvent Methods

FieldActivatedEvent uses the following methods:

### getActivatedFunctionCodeId()

Returns the newly activated field's FunctionCode.

### getStateId()

Returns the state identifier that contains the function code corresponding to the newly activated field.

### getCursorPosition()

Returns the cursor position in the newly activated field. If the operator used right and left arrow function keys, the cursor position might not be at the end of the data in the field. The GUI might use the cursor position to set the caret position of a text field.

### getActivatedFunctionCodeId()

This method returns the function code identifier corresponding to the newly activated field. This id can be used to retrieve the actual FunctionCode instance by using the FunctionCodeList.getFunctionCodeWithId(int) method. The FunctionCodeList can be retrieved by using the State.getValidFullScreenFunctionCodes() method.

```
public int getActivatedFunctionCodeId()
```

**Returns:** The function code id for the newly activated field.

### getStateId()

This method returns the state identifier that contains the function corresponding to the newly activated field.

```
public int getStateId()
```

**Returns:** The state id that contains the function code id for the newly activated field.

### getCursorPosition()

This method returns the cursor position for the newly activated field. The cursor position is the point in the field at which all new input is inserted.

```
public int getCursorPosition()
```

**Returns:** The cursor position in the newly activated field.

---

## Class FieldDeactivatedEvent

Instances of this class are used by the Java IO Processor when processing a full-screen state. When a field is activated, the JIOP raises a `FieldDeactivatedEvent` for the previously active field (if any). The GUI can respond by disabling a corresponding GUI component.

`FullScreenFieldListeners` can receive `FieldDeactivatedEvents`.

### FieldDeactivatedEvent Methods

`FieldDeactivatedEvent` uses the following methods:

#### **getDeactivatedFunctionCodeId()**

Returns the newly deactivated field's `FunctionCode`.

#### **getStateId()**

Returns the state identifier that contains the function code corresponding to the newly deactivated field.

#### **getDeactivatedFunctionCodeId()**

This method returns the function code identifier corresponding to the newly deactivated field. This id can be used to retrieve the actual `FunctionCode` instance by using the `FunctionCodeList.getFunctionCodeWithId(int)` method. The `FunctionCodeList` can be retrieved by using the `State.getValidFullScreenFunctionCodes()` method.

```
public int getDeactivatedFunctionCodeId()
```

**Returns:** The function code id for the newly deactivated field.

#### **getStateId()**

This method returns the state identifier that contains the function corresponding to the newly deactivated field.

```
public int getStateId()
```

**Returns:** The state id that contains the function code id for the newly deactivated field.

---

## Class FieldDataUpdatedEvent

An input field can be modified in one of the following ways:

- Keystrokes cause data to be inserted into the field.
- Backspace or delete keys cause data to be removed from the field.
- The field can be cleared.
- The left or right arrow keys cause the cursor position to change.
- The input sequence can be cleared.
- WRITE or PUTLONG requests from the application.

The JIOP raises `FieldDataUpdatedEvents` to indicate changes to the input field caused by any of the above events.

`FullScreenFieldListeners` can receive `FieldDataUpdatedEvents` related to full-screen input fields. `InputFieldListeners` can receive `FieldDataUpdatedEvents` related to non-full-screen input fields.

### FieldDataUpdatedEvent Methods

`FieldDataUpdatedEvents` uses the following methods:

#### **getFunctionCode()**

Returns the `FunctionCode` instance for the field that has been updated.

**getFunctionCodeId()**

Returns the function code identifier for the field that has been updated.

**getStateId()**

Returns the state identifier of the state that contains the function code of the field that has been updated.

**isFullScreenField()**

Indicates whether the field that has been updated is a full-screen field.

**getUpdatedData()**

Returns the new contents of the field.

**getCursorPosition()**

Returns the new cursor position within the field.

**getChangeSource()**

Indicates the cause of the update.

**getDoNotInitializeScreenData()**

Indicates that a CBASIC application WRITE/PUTLONG combination updated the field.

**getFunctionCode()**

This method returns the function code instance for the field that has been updated.

```
public FunctionCode getFunctionCode()
```

**Returns:** The function code for the field that has been updated.

**getFunctionCodeId()**

This method returns the function code identifier for the field that has been updated.

```
public int getFunctionCodeId()
```

**Returns:** The function code identifier for the field that has been updated.

**getStateId()**

This method returns the state identifier that contains the function code for the field that has been updated.

```
public int getStateId()
```

**Returns:** The state id that contains the function code id for the field that has been updated.

**isFullScreenField()**

This method indicates whether the field that has been modified is a full-screen field.

```
public boolean isFullScreenField()
```

**Returns:** True, if the updated field is a full-screen field.

**getUpdatedData()**

This method returns the new contents of the input field that has been updated.

```
public String getUpdatedData()
```

**Returns:** A String containing the new contents of the input field.

**getCursorPosition()**

This method returns the cursor position for the updated field. The cursor position is the point in the field at which all new input is inserted.

```
public int getCursorPosition()
```

**Returns:** The cursor position in the updated field.

### **getChangeSource()**

This method returns the source of the update to the input field. For example, the source might be the keyboard, the scanner, or a CBASIC WRITE/PUTLONG combination. The valid values for the change source are constants defined in IOPInputQueue.

```
public int getChangeSource()
```

**Returns:** A constant from IOPInputQueue representing the source of the field change.

### **getDoNotInitializeScreenData()**

This method indicates that a change has been made to the input field through a CBASIC application WRITE/PUTLONG combination, but that no GUI updates should be made.

```
public boolean getDoNotInitializeScreenData()
```

**Returns:** True, if the GUI should not be updated.

---

## **Class FullScreenAttributes**

The Input Sequence Table Utility allows configuring attributes related to a full-screen application. This class allows access to these attributes. The Enhanced Full Screen State Table defines three display areas:

- The display as keyed (DAK) area. This is the display area where keyboard input is displayed as it is keyed.
- The error area. This is the display area where the IO processor displays errors.
- The prompt area. This is the display area where the IO processor displays prompts.

Each of these areas has video attributes that can be configured in the input sequence table. The FullScreenAttributes class allows access to these attributes through a nested class called VideoParms described below.

## **FullScreenAttributes Methods**

FullScreenAttributes uses the following methods:

### **getVideoParms(int)**

Returns a VideoParms instance for one of the three display areas (DAK, error, or prompt).

### **getPromptAreaVideoParms()**

Returns a VideoParms instance for the prompt area of the full-screen display.

### **getErrorAreaVideoParms()**

Returns a VideoParms instance for the error area of the full-screen display.

### **getDakAreaVideoParms()**

Returns a VideoParms instance for the display as keyed (DAK) area of the full-screen display.

### **getTabRightFCode()**

The State Table Utility allows the definition of a function code to be treated as a tab key (in addition to the regular tab key on an ANPOS keyboard). This method returns the function code of the additional forward tab key.

### **getTabLeftFCode()**

The State Table Utility allows the definition of a function code to be treated as a back tab key (in addition to the regular shift-tab key on an ANPOS keyboard). This method returns the function code of the additional back tab key.

### **getNumberOfPositions()**

Returns the number of numeric positions to the right of the decimal point as defined in the full-screen state table. Note that the JIOP does not support “display with editing”. The JIOP ignores this value, but the Java GUI can choose to use it to format numeric input.

### **getQuoteSubstitute()**

Full-Screen input sequences can contain alphanumeric data in the input sequence buckets that

are delivered to the CBASIC application. The fields in the input sequence are delimited with quotation characters. The State Table Utility allows a quote substitution character to be defined for full-screen state tables. The substitute quote character replaces any quotation character that is inserted into the sequence. The CBASIC application can then reverse the substitution to restore the original data values. This method returns the substitute quotation character as defined in the state table.

### **getVideoParms(int)**

Returns a VideoParms instance for one of the three display areas (DAK, error, or prompt).

```
public VideoParms getVideoParms(int area)
```

#### **Parameters:**

**area** An integer value representing the display area to retrieve. Valid values are: JIOProcessor.DAK, JIOProcessor.ERROR, and JIOProcessor.PROMPT.

**Returns:** The VideoParms instance for the requested display area.

### **getPromptAreaVideoParms()**

Returns the VideoParms instance for the prompt display area as defined in a full-screen state table.

```
public VideoParms getPromptAreaVideoParms()
```

**Returns:** The VideoParms instance for the prompt display area.

### **getErrorAreaVideoParms()**

Returns the VideoParms instance for the error display area as defined in a full-screen state table.

```
public VideoParms getErrorAreaVideoParms()
```

**Returns:** The VideoParms instance for the error display area.

### **getDakAreaVideoParms()**

Returns the VideoParms instance for the display as keyed (DAK) display area as defined in a full-screen state table.

```
public VideoParms getDakAreaVideoParms()
```

**Returns:** The VideoParms instance for the display as keyed (DAK) display area.

### **getTabRightFCode()**

This method returns the function code of the additional forward tab key.

```
public short getTabRightFCode()
```

**Returns:** The function code value that has been defined as an additional forward tab key.

### **getTabLeftFCode()**

This method returns the function code of the additional backward tab key.

```
public short getTabLeftFCode()
```

**Returns:** The function code value that has been defined as an additional backward tab key.

### **getNumberOfPositions()**

Returns the number of numeric positions to the right of the decimal point as defined in the full-screen state table.

```
public short getNumberOfPositions()
```

**Returns:** The number of positions to the right of the decimal point.

### **getQuoteSubstitute()**

This method returns the substitute quotation character as defined in the full-screen state table.

```
public char getQuoteSubstitute()
```

**Returns:** The character defined to replace the quotation character when inserted into an input sequence.

---

## **Class IOPReadyListener**

This class is the listener interface for IOP ready events. After the Java I/O processor (JIOP) is constructed, it raises a series of status events that end with the IOPReadyEvent. At that time, the JIOP is ready for normal operation.

### **IOPReadyListener Methods**

IOPReadyListener uses the following methods:

#### **iopReady(IOPReadyEvent)**

Called when the I/O processor is ready, connected to the application.

#### **iopStatus(IOPInitStatus)**

Informs the listener of the progress of the JIOP initialization.

#### **iopReady(IOPReadyEvent)**

This method is called when the I/O processor is ready, connected to the application

```
public abstract void iopReady(IOPReadyEvent iopr)
```

#### **Parameters:**

**iopr** Ready event

#### **iopStatus(IOPInitStatus)**

This method informs the listener of the progress of the JIOP initialization.

```
public abstract void iopStatus(IOPInitStatus iops)
```

#### **Parameters:**

**iops** Status event



---

## Class **IOPReadyEvent**

This event indicates that the JIOP has loaded required data and connected to the application. Prior to this event, the JIOP is not in a valid state. Applications should not call methods on the JIOP until this event is received.

This class is an extension of the `EventObject` class.

---

## Class IOInitStatus

This event is raised as the Java I/O processor initializes to inform the listener of progress

### IOInitStatus Method

IOInitStatus uses one method:

#### **getStatus**

Returns the string describing the status. The status messages indicate the activities that occur as the JIOP initializes. The activities include loading tables and opening devices.

#### **getStatus**

This method returns the string describing the status.

```
public string getStatus()
```

**Returns:** Status string

---

## Class PromptChangeListener

This method is the listener interface for receiving prompt change events. During operation, the Java I/O processor (JIOP) requires prompt changes to occur on the operator 2x20 display. This event allows the user interface to monitor the 2x20 prompts that the JIOP displays and to put the text in an appropriate 2x20 control.

PromptChangeListener instances receive only prompt updates from the Java IO processor. These updates include prompts and errors as defined in the state table, as well as the keyboard input.

PromptChangeListeners **do not** receive display updates from the CBASIC or C application. The Java GUI must implement an ANDisplayHandler to receive these updates. See “Class ANDisplayHandler” on page 548.

### PromptChangeListener Method

PromptChangeListener uses the following method:

#### **promptChanged(PromptChangeEvent)**

Called when the I/O processor prompt changes.

#### **promptChanged(PromptChangeEvent)**

This method is called when the I/O processor prompt changes.

**Note:** This is a special Java GUI consideration for full-screen state prompt changes when the display area is defined as a 2x20 area, rather than a single line. The PromptChangeEvent determines the starting row and column for the prompt, but the GUI is responsible for breaking the prompt data into two separate lines for display.

As an example, assume the state table defines separate 2x20 display areas for the prompt area, the display as keyed area and the error area as shown in Figure 18 on page 606:

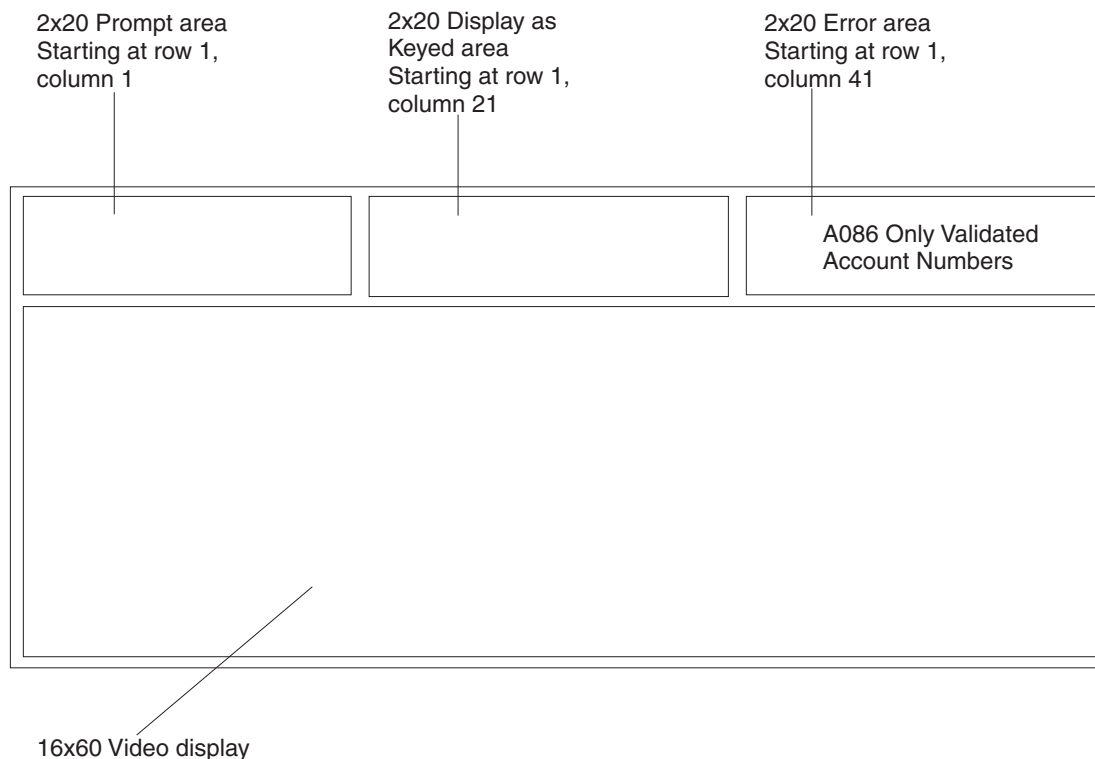


Figure 18. 2x20 Display Areas Example

Now suppose that the GUI receives a `PromptChangeEvent`. The starting row is 1, and the starting column is 41. The prompt data is "A083 Only Validated Account Numbers". The first step is to determine if the prompt update occurs in one of the three full-screen display areas. To accomplish this, check if the `PromptChangeEvent` starting row and column occurs inside of one of the rectangles defining the prompt area, the display as keyed area, or the error area. These rectangles can be computed by using the `VideoParms.getRow()`, `VideoParms.getColumn()`, `VideoParms.is2x20()`, and `VideoParms.getWidth()` methods. The `VideoParms` for each of the three display areas can be retrieved by using methods of the `FullScreenAttributes` class. To obtain an instance of the `FullScreenAttributes`, the `getFullScreenAttributes` method defined in the `StateTableProcessor` class must be invoked. An instance of the `StateTableProcessor` can be obtained by invoking the `getTableProcessor` static method defined in the `JIOProcessor` class.

For this example, the prompt change falls within the area defining the error area. Because the error area is defined as a 2x20 display area, the prompt "A083 Only Validated Account Numbers" would be split and displayed on two separate lines if the Java GUI honored the 2x20 configuration of the error display area.

From this example, you can see that it is the responsibility of the Java GUI to determine if prompt updates are being written inside any of the three display areas. The Java GUI might choose to apply formatting based on the configuration of the appropriate display area as defined in the state table.

```
public abstract void promptChanged(PromptChangeEvent pce)
```

#### Parameters:

**pce** Prompt change event

---

## Class PromptChangeEvent

This class is an event that indicates that a 2x20 display prompt change has occurred within the I/O processor.

### PromptChangeEvent Methods

PromptChangeEvent uses the following methods:

#### **getColumn()**

Returns the column to display the prompt on.

#### **getPrompt()**

Returns the prompt.

#### **getRow()**

Returns the row to display the prompt on.

#### **getColumn()**

Returns the column to display the prompt on.

```
public int getColumn()
```

**Returns:** Column

#### **getPrompt()**

Returns the current prompt.

```
public String getPrompt()
```

**Returns:** Prompt

#### **getRow()**

Returns the row to display the prompt on.

```
public int getRow()
```

**Returns:** Prompt

---

## Class IOPShutdownListener

This interface allows the caller to listen for shutdown (close) of the Java I/O processor. Shutdown indicates that the application has ended. The user interface should provide an appropriate response.

### IOPShutdownListener Method

IOPShutdownListener uses the following method:

#### **jiopShutdown()**

Called when the JIOP closes.

#### **jiopShutdown()**

This method is called when the JIOP closes.

```
public abstract void jiopShutdown()
```

---

## Class SystemBusyListener

This interface enables listening for System Busy to occur in the JIOP.

### SystemBusyListener Method

SystemBusyListener uses the following method:

#### **systemBusy(SystemBusyEvent)**

Called when the JIOP enters System Busy state.

#### **systemBusy(SystemBusyEvent)**

This method is called when the JIOP enters System Busy state.

```
public abstract void systemBusy(SystemBusyEvent sbe)
```

#### **Parameters:**

**sbe**     System busy event

---

## Class SystemBusyEvent

If the JIOP receives input while the InputQueue is locked, JIOP might enter the system busy state. In this state, the system busy message configured in the input state table is displayed. Either the operator keying the system busy clear key or the POS application unlocking the JIOP can clear the system busy state.

### SystemBusyEvent Method

SystemBusyEvent uses the following method:

#### **getSystemBusyClearKey()**

Queries the key to clear System Busy.

#### **getSystemBusyClearKey()**

This method is called to query the key to clear System Busy.

```
public int getSystemBusyClearKey()
```

**Returns:** Gear function code



---

## Class IOInputQueue

This class is the interface for the input queue for the I/O processor. This class provides the input mechanism for sending data and function codes to the I/O processor. There is only one input queue present in the system.

### IOInputQueue Methods

IOInputQueue uses the following methods:

#### **addLabelInputListener(LabelInputListener)**

Adds a LabelInput listener

#### **addPOSKeyListener(POSKeyListener)**

Adds a POS key listener. This method is deprecated. Use `removePOSKeyListenerEx(POSKeyListenerEx)` instead.

#### **addPOSKeyListenerEx(POSKeyListenerEx)**

Adds an extended POS key listener.

#### **addQueueStatusListener(QueueStatusChangeListener)**

Adds a queue status listener

#### **getQueue()**

Returns the IOInputQueue in the system.

#### **isLocked()**

Check if the queue is locked

#### **isPurged()**

Query if the I/O processor is locked and was purged on the lock

#### **postChar(char)**

Add a char to the queue

#### **postCommand(int)**

Add a function code to the input queue

#### **postString(String)**

Add a String to the queue

#### **removeLabelInputListener(LabelInputListener)**

Removes a LabelInput listener

#### **removePOSKeyListener(POSKeyListener)**

Removes a POS key listener. This method is deprecated. Use `removePOSKeyListenerEx(POSKeyListenerEx)` instead.

#### **removePOSKeyListenerEx(POSKeyListenerEx)**

Removes an extended POS key listener.

#### **removeQueueStatusListener(QueueStatusChangeListener)**

Removes a queue status listener

#### **addLabelInputListener(LabelInputListener)**

Adds a LabelInput listener

```
public synchronized void addLabelInputListener(LabelInputListener listener)
```

#### **Parameters:**

##### **listener**

Label listener

### **addPOSKeyListener(POSKeyListener)**

Adds a POS key listener. This method is deprecated. Use `addPOSKeyListenerEX(POSKeyListenerEx)` instead.

```
public synchronized void addPOSKeyListener(POSKeyListener listener)
```

#### **Parameters:**

##### **listener**

Key listener

### **addPOSKeyListenerEx(POSKeyListenerEx)**

Adds an extended POS key listener.

```
public synchronized void addPOSKeyListenerEx(POSKeyListenerEx listener)
```

#### **Parameters:**

##### **listener**

Key listener

### **addQueueStatusListener (QueueStatusChangeListener)**

Adds a queue status listener

```
public synchronized void addQueueStatusListener(QueueStatusChangeListener q)
```

#### **Parameters:**

**q** Queue status change listener

### **getQueue()**

Returns the `IOPInputQueue` in the system.

```
public static IOPInputQueue getQueue()
```

**Returns:** `IOPInputQueue` the queue

### **isLocked()**

Check if the queue is locked

```
public boolean isLocked()
```

**Returns:** True if the input queue is locked

### **isPurged()**

Query if the I/O processor is locked and was purged on the lock.

```
public boolean isPurged()
```

**Returns:** True if the queue was purged on the previous lock.

### **postChar(char)**

Add a char to the queue

```
public synchronized void postChar(char c) throws QueueLockedException,  
    StateForbidsInputException
```

#### **Parameters:**

**c** The char to add

#### **Throws:**

##### **QueueLockedException**

If the `IOPInputQueue` is locked

**StateForbidsInputException**

If input disallowed

**postChar(char, int)**

Add a character to the input queue.

```
public synchronized void postChar(char c, int src) throws QueueLockedException,  
    StateForbidsInputException
```

**Parameters:**

**c** The character to add

**src** Source of the input, as defined in “IOInputQueue Public Data” on page 619.

**Note:** The Java GUI might call this version of the `postChar()` method and pass `IOInputQueue.EXTERNALGUI` as the `src` parameter. Optionally, the Java GUI might also call the previous version of the `postChar()` method without indicating the source. Either method is acceptable.

**postCommand(int)**

Add a function code to the input queue

```
public synchronized void postCommand(int f) throws QueueLockedException,  
    StateForbidsInputException
```

**Parameters:**

**f** Function code

**Throws:****QueueLockedException**

If the `IOInputQueue` is locked

**StateForbidsInputException**

If input is disallowed

**postCommand(int, int)**

Add a function code to the input queue

```
public synchronized void postCommand(int f, int src) throws QueueLockedException,  
    StateForbidsInputException
```

**Parameters:**

**f** Function code

**src** Source of the input, as defined in “IOInputQueue Public Data” on page 619.

**Note:** The Java GUI might call this version of the `postCommand()` method and pass `IOInputQueue.EXTERNALGUI` as the `src` parameter. Optionally, the Java GUI might also call the previous version of the `postCommand()` method without indicating the source. Either method is acceptable.

**postString(String)**

Add a string to the queue.

```
public synchronized void postString(string s) throws QueueLockedException,  
    StateForbidsInputException
```

**Parameters:**

**s** The string to add

**Throws:**

**QueueLockedException**

If the `IOPInputQueue` is locked

**StateForbidsInputException**

If input disallowed

**postString(String, int)**

Add a string to the queue.

```
public synchronized void postString(String s, int src) throws QueueLockedException,
StateForbidsInputException
```

**Parameters:**

**s**        The string to add

**src**      Source of the input, as defined in “`IOPInputQueue Public Data`” on page 619.

**Note:** The Java GUI might call this version of the method and pass `IOPInputQueue.EXTERNALGUI` as the `src` parameter. Optionally, the Java GUI might also call the previous version of the `postString()` method without indicating the source. Either method is acceptable.

**removeLabelInputListener (LabelInputListener)**

Removes a `LabelInput` listener

```
public synchronized void removeLabelInputListener(LabelInputListener
listener)
```

**Parameters:**

**listener**        Label listener

**removePOSKeyListener (POSKeyListener)**

Removes a POS key listener. This method is deprecated. Use `removePOSKeyListenerEx(POSKeyListenerEx)` instead.

```
public synchronized void removePOSKeyListener(POSKeyListener listener)
```

**Parameters:**

**listener**        Key listener

**removePOSKeyListenerEx (POSKeyListenerEx)**

Removes an extendedPOS key listener.

```
public synchronized void removePOSKeyListenerEx(POSKeyListenerEx listener)
```

**Parameters:**

**listener**        Key listener

**removeQueueStatusListener (QueueStatusChangeListener)**

Removes a queue status listener

```
public synchronized void removeQueueStatusListener(QueueStatusChangeListener q)
```

**Parameters:**

**q**        Queue status change listener

---

## Class QueueChangeListener

This class is the listener interface for receiving queue status change events. The input queue for the Java I/O processor locks and unlocks due to application input and input state table information. When the lock/unlock occurs, all objects subscribed to QueueChangeListener are notified.

### QueueChangeListener Method

QueueChangeListener uses the following method:

#### **queueStatusChange(QueueStatusChangeEvent)**

Called when the input queue locks and unlocks.

#### **queueStatusChange (QueueStatusChangeEvent)**

This method is called when the input queue locks and unlocks.

```
public abstract void queueStatusChange(QueueStatusChangeEvent q)
```

#### **Parameters:**

**q**      Event

---

## Class QueueStatusChangeEvent

This class is fired when the IOPInputQueue lock status changes. The IOPInputQueue locks and unlocks according to input state table information and application activity.

### QueueStatusChangeEvent Method

QueueStatusChangeEvent uses the following method:

#### **isLocked()**

Indicates if the queue is locked.

#### **isLocked()**

Indicates if the queue is locked.

```
public boolean isLocked()
```

**Returns:** True if queue is locked, false otherwise.

---

## Class VideoParms

This class is used to encapsulate all the attribute information of each of the three video display areas (the display as keyed area, the prompt area, and the error area). The attributes of each of these display areas is defined in the common area of a full-screen state table using the Input Sequence Table Utility.

The VideoParms class is a nested class and is defined by the FullScreenAttributes class. Therefore, method calls on instances of this class must scope the method call to the FullScreenAttributes class.

### VideoParms Method

VideoParms uses the following methods:

#### **getFlags()**

Returns a bit set indicating if the display area is 2x20, has a border, and is centered.

#### **hasBorder()**

Indicates if the display area has been defined with a surrounding border.

#### **isCentered()**

Indicates if the display area has been defined to center the data within the input field.

#### **is2x20()**

Indicates if the display area is configured as a 2 rows by 20 columns, or if it is a single row.

#### **getRow()**

Returns the starting row on the video display for the display area represented by the VideoParms instance.

#### **getColumn()**

Returns the starting column on the video display for the display area represented by the VideoParms instance.

#### **getWidth()**

Returns the maximum number of characters that the display area can display.

#### **getAttributes()**

Returns the color attributes of this display area. This value is not used by the JIOP, but is available for use by the Java GUI, if required.

#### **getTopLeftBorderChar()**

Returns the ASCII character defined as the top left border character for this display area. This value is not used by the JIOP, but is available for use by the Java GUI, if required.

**getTopBorderChar()**

Returns the ASCII character defined as the top border character for this display area. This value is not used by the JIOP, but is available for use by the Java GUI, if required.

**getTopRightBorderChar()**

Returns the ASCII character defined as the top right border character for this display area. This value is not used by the JIOP, but is available for use by the Java GUI, if required.

**getLeftBorderChar()**

Returns the ASCII character defined as the left border character for this display area. This value is not used by the JIOP, but is available for use by the Java GUI, if required.

**getRightBorderChar()**

Returns the ASCII character defined as the right border character for this display area. This value is not used by the JIOP, but is available for use by the Java GUI, if required.

**getBottomLeftBorderChar()**

Returns the ASCII character defined as the bottom left border character for this display area. This value is not used by the JIOP, but is available for use by the Java GUI, if required.

**getBottomBorderChar()**

Returns the ASCII character defined as the bottom border character for this display area. This value is not used by the JIOP, but is available for use by the Java GUI, if required.

**getBottomRightBorderChar()**

Returns the ASCII character defined as the bottom right border character for this display area. This value is not used by the JIOP, but is available for use by the Java GUI, if required.

**getFlags()**

Returns a bit set indicating if the display area is 2x20, has a border, and is centered. There are also individual accessor methods to determine the status of each of these bits.

```
public int getFlags()
```

**Returns:** Configuration bits for the display area.

**hasBorder()**

Indicates if the display area has been defined with a surrounding border.

```
public boolean hasBorder()
```

**Returns:** True, if the display area is defined with a border.

**isCentered()**

Indicates if the display area has been defined to center the data with the input field.

```
public boolean isCentered()
```

**Returns:** True, if the data is to be centered in the display area.

**is2x20()**

Indicates if the display area is configured as a 2 rows by 20 columns, or if it is a single line.

```
public boolean is2x20()
```

**Returns:** True, if the display area is 2 rows by 20 columns, false if the display area is a single line.

**getRow()**

Returns the starting row on the video display for the display area represented by the VideoParms instance.

```
public short getRow()
```

**Returns:** The starting row for the display area (not including any border).

### **getColumn()**

Returns the starting column on the video display for the display area represented by the VideoParms instance.

```
public short getColumn()
```

**Returns:** The starting column for the display area (not including any border).

### **getWidth()**

Returns the maximum number of characters that can be displayed in the display area represented by the VideoParms instance.

```
public short getWidth()
```

**Returns:** The character width of the display area (not including any border).

### **getAttributes()**

Returns the color attributes instance for the display area represented by the VideoParms instance.

```
public Attributes getAttributes()
```

**Returns:** The color attributes instance for the display area.

### **getTopLeftBorderChar()**

Returns the ASCII character defined as the top left border character for this display area. This value is not used by the JIOP, but is available for use by the Java GUI, if required.

```
public char getTopLeftBorderChar()
```

**Returns:** The ASCII character defined as the top left border character.

### **getTopBorderChar()**

Returns the ASCII character defined as the top border character for this display area. This value is not used by the JIOP, but is available for use by the Java GUI, if required.

```
public char getTopBorderChar()
```

**Returns:** The ASCII character defined as the top border character.

### **getTopRightBorderChar()**

Returns the ASCII character defined as the top right border character for this display area. This value is not used by the JIOP, but is available for use by the Java GUI, if required.

```
public char getTopRightBorderChar()
```

**Returns:** The ASCII character defined as the top right border character.

### **getLeftBorderChar()**

Returns the ASCII character defined as the left border character for this display area. This value is not used by the JIOP, but is available for use by the Java GUI, if required.

```
public char getLeftBorderChar()
```

**Returns:** The ASCII character defined as the left border character.

### **getRightBorderChar()**

Returns the ASCII character defined as the right border character for this display area. This value is not used by the JIOP, but is available for use by the Java GUI, if required.

```
public char getRightBorderChar()
```

**Returns:** The ASCII character defined as the right border character.



### **getBottomLeftBorderChar()**

Returns the ASCII character defined as the bottom left border character for this display area. This value is not used by the JIOP, but is available for use by the Java GUI, if required.

```
public char getBottomLeftBorderChar()
```

**Returns:** The ASCII character defined as the bottom left border character.

### **getBottomBorderChar()**

Returns the ASCII character defined as the bottom border character for this display area. This value is not used by the JIOP, but is available for use by the Java GUI, if required.

```
public char getBottomBorderChar()
```

**Returns:** The ASCII character defined as the bottom border character.

### **getBottomRightBorderChar()**

Returns the ASCII character defined as the bottom right border character for this display area. This value is not used by the JIOP, but is available for use by the Java GUI, if required.

```
public char getBottomRightBorderChar()
```

**Returns:** The ASCII character defined as the bottom right border character.

---

## **IOInputQueue Public Data**

The following are public data defined in the IOInputQueue class. These constants are used to indicate the source of input field changes.

```
public static int UNKNOWN = 0;
```

The source of the input field change is unknown.

```
public static int KEYBOARD = 1;
```

The input field was updated due to keyboard input.

```
public static int AUTOEOF = 2;
```

The input field is defined to automatically generate a function code when the maximum length has been reached. This value indicates the input field has been modified by automatic end of field processing.

```
public static int SCANNER = 3;
```

The input field was updated due to scanner input.

```
public static int WRITEPUTLONG = 4;
```

The input field was updated due to a CBASIC application WRITE/PUTLONG combination.

```
public static int CLEARFIELD = 5;
```

The input field was cleared.

```
public static int CLEARSEQUENCE = 6;
```

The entire input sequence was cleared.

```
public static int AUTOTAB = 7;
```

The input field is a full-screen input field defined to automatically generate a tab when the maximum input length has been reached.

```
public static int KEYEDLABEL = 8;
```

The input field has been modified due to a label being keyed on the keyboard.

```
public static int EXTERNALGUI = 9;
```

The input field has been modified due to an external API call to post data into the input queue.

---

## Interface InputFieldListener

Objects that implement the `InputFieldListener` interface can receive events indicating that an input field has been updated. Input field updates can include adding data, removing data, or changing the cursor position. The JIOP raises `FieldDataUpdatedEvents` to indicate changes to the input field caused by changes to the field data, or changes to the cursor position.

`FullScreenFieldListeners` can receive `FieldDataUpdatedEvents` related to full-screen input fields. `InputFieldListeners` can receive `FieldDataUpdatedEvents` related to non-full-screen input fields.

An `InputFieldListener` is registered by calling the `StateTableProcessor.addInputFieldListener()` method.

## InputFieldListener Methods

`InputFieldListener` uses the following methods:

### **fieldDataUpdated(FieldDataUpdatedEvent)**

This method is called when the input field data has been modified, or the cursor position has changed.

### **fieldDataUpdated(FieldDataUpdatedEvent)**

This method is called when the input field data has been modified, or the cursor position has changed.

**Note:** This method is called on a JIOP thread. If the Java GUI must update AWT or Swing components while processing this method, it should utilize the `SwingUtilities.invokeLater()` method to cause a context switch to the AWT thread.

```
public void fieldDataUpdated(FieldDataUpdatedEvent event)
```

### **Parameters:**

**event** An event encapsulating the data and cursor position after the field has been modified.

---

## Interface FullScreenFieldListener

The `FullScreenFieldListener` interface extends the `InputFieldListener` interface. The `FullScreenFieldListener` can be notified of the field data and cursor position changes through the `fieldDataUpdated(FieldDataUpdatedEvent)` method.

A full-screen state includes the concept of an “active” field. The active field is the full-screen input field that receives keyboard or scanner input. There can only be one active field at a time. The `FullScreenFieldListener` interface includes methods for notifying fields becoming activated or deactivated.

The JIOP activates an input field when unlocking into a full-screen state. If an input field is active when a different input field is activated, the previously active input field is deactivated.

An `FullScreenFieldListener` is registered by calling the `StateTableProcessor.addFullScreenFieldListener()` method.

## FullScreenFieldListener Methods

`FullScreenFieldListener` uses the following methods:

### **fieldActivated(FieldActivatedEvent)**

This method is called when the JIOP activates an input field.

### **fieldDeactivated(FieldDeactivatedEvent)**

This method is called when the JIOP deactivates an input field.

### fieldActivated(FieldActivatedEvent)

This method is called when the JIOP activates an input field. If the Java GUI must update AWT or Swing components while processing this method, it should utilize the `SwingUtilities.invokeLater()` method to cause the context switch to the AWT thread.

```
public void fieldActivated(FieldActivatedEvent event)
```

#### Parameters:

**event** An event encapsulating the field activation information.

### fieldDeactivated(FieldDeactivatedEvent)

This method is called when the JIOP deactivates an input field. If the Java GUI must update AWT or Swing components while processing this method, it should utilize the `SwingUtilities.invokeLater()` method to cause the context switch to the AWT thread.

```
public void fieldDeactivated(FieldDeactivatedEvent event)
```

#### Parameters:

**event** An event encapsulating the field activation information.

---

## Interface POSKeyListener

**Note:** This interface has been deprecated. Use the `POSKeyListenerEx` interface instead.

This interface allows the user interface to listen for data and function code input from the POS keyboard. The user interface can display the input in an appropriate control and do input validation as required.

## POSKeyListener Methods

`POSKeyListener` uses the following methods:

### dataKeyPress(char)

Listen for data key events with the option of discontinuing processing of the key.

### functionKeyPress(int)

Listen for function key events with the option of discontinuing processing of the key.

### dataKeyPress(char)

This method listens for data key events with the option of discontinuing processing of the key. ***This method should be implemented efficiently to ensure quick response to keyboard activity.***

```
public abstract boolean dataKeyPress(char data)
```

#### Parameters:

**data** The data from the keyboard.

**Returns:** True to discontinue processing, false to continue.

### functionKeyPress(int)

This method listens for function key events with the option of discontinuing processing of the key. ***This method should be implemented efficiently to ensure quick response to keyboard activity.***

```
public abstract boolean functionKeyPress(int code)
```

#### Parameters:

**code** The function code from the keyboard.

**Returns:** True to discontinue processing, false to continue.

---

## Interface POSKeyListenerEx

This interface allows the user interface to listen for data and function code input from the POS keyboard. The user interface can display the input in an appropriate control and do input validation as required.

### POSKeyListenerEx Methods

POSKeyListenerEx uses the following methods:

#### dataKeyPress(String)

Listen for data key events with the option of discontinuing processing of the key. The string contains a single character corresponding to the value of the key that was pressed. In the case of keyboard mappings that have a single key stem defined as a double or triple zero ("00" or "000"), the string passed to this method also contains a double or triple zero instead of a single character.

**Note:** The Enhanced Full Screen State Table allows keyboard input corresponding to a specific function code to be automatically converted to uppercase. The string passed in this method is **not** converted to uppercase, allowing the Java GUI to determine the actual input character prior to being converted to uppercase for insertion into the input sequence.

#### functionKeyPress(int)

Listen for function key events with the option of discontinuing processing of the key.

#### dataKeyPress(String)

This method listens for data key events with the option of discontinuing processing of the key. ***This method should be implemented efficiently to ensure quick response to keyboard activity. AWT or Swing components should not be modified without utilizing SwingUtilities.invokeLater() to cause a context switch from the JIOP thread to the AWT thread.***

```
public abstract boolean dataKeyPress(String data)
```

#### Parameters:

**data** The data from the keyboard.

#### functionKeyPress(int)

This method listens for function key events with the option of discontinuing processing of the key. ***This method should be implemented efficiently to ensure quick response to keyboard activity. AWT or Swing components should not be modified without utilizing SwingUtilities.invokeLater() to cause a context switch from the JIOP thread to the AWT thread.***

```
public abstract boolean functionKeyPress(int data)
```

#### Parameters:

**code** The function code from the keyboard.

**Returns:** True to discontinue processing; false to continue.

---

## Interface LabelInputListener

Allows the user interface to listen to bar label input from the scanner and keyboard.

### LabelInputListener Methods

POSKeyListener uses the following method. ***AWT or Swing components should not be modified without utilizing `SwingUtilities.invokeLater()` to cause a context switch from the JIOP thread to the AWT thread.***

#### **invalidKeyedLabel(string)**

Informs the listener that the operator has entered an invalid label.

#### **labelInput(LabelInputEvent)**

Informs the listener of the label input.

#### **invalidKeyedLabel(string)**

Informs the listener that the operator has entered an invalid label. Labels can be flagged as invalid for a variety of reasons including modulo check failure.

```
public abstract void invalidKeyedLabel(String label)
```

#### **Parameters:**

**label**    Label entered by operator

#### **labelInput(LabelInputEvent)**

Informs the listener about the label input. For a single label, the listener is notified of the label composition as described in the Label Format Table.

**Note:** A single bar label can be composed into multiple functionCode/data pairs

```
public abstract void labelInput(LabelInputevent li)
```

#### **Parameters:**

**li**        Label input event.

---

## Class FunctionCodeList

This class is a list of all the valid `FunctionCode` objects for a full-screen state. `FunctionCodeList` extends the `JDK Hashtable` class. Each `FunctionCode` uses the tab order as a key in the hashtable.

`FunctionCodes` within a state are persistent objects. Therefore, it is important to use the `State.getValidFullScreenFunctionCodes()` to gain access to a full-screen state `FunctionCode`, rather than use the `State.getValidFunctionCodes()` method.

### FunctionCodeList Methods

Besides the usual `get()` and `put()` methods of the `Hashtable` class, `FunctionCodeList` uses the following methods:

#### **getFirstFunctionCode()**

Returns the function code with the lowest tab order.

#### **getFunctionCodeWithTabOrder(int)**

Returns the function code with the requested tab order.

#### **getFunctionCodeWithId(int)**

Returns the function code with the requested id.

#### **getPreviousFunctionCode(FunctionCode)**

Given a function code, returns the function code with the next lowest tab order.

#### **getNextFunctionCodeOrder(int)**

Given a function code, returns the function code with the next highest tab order.

#### **getCurrentFunctionCode()**

Returns the function code of the currently active input field.

#### **getTabOrderOfCurrent()**

Returns the tab order of the currently active input field.

#### **getStateId()**

Returns the id of the full-screen state containing the `FunctionCodeList` instance.

#### **getLowestTabOrder()**

Returns the tab order of the function code with the lowest tab order in the list.

#### **clearInputFields(boolean)**

Clears the input fields associated with each of the function codes in the list.

#### **getFirstFunctionCode()**

This method returns the function code with the lowest tab order.

```
public FunctionCode getFirstFunctionCode()
```

**Returns:** `FunctionCode` in the hashtable with the lowest tab order. This is the function code with the lowest tab order in the full-screen state.

#### **getFunctionCodeWithTabOrder(int)**

This method returns the function code with the requested tab order.

```
public FunctionCode getFunctionCodeWithTabOrder(int order)
```

#### **Parameters:**

**order** Tab order of requested function code.

**Returns:** `FunctionCode` in the hashtable with the requested tab order, or null if function code with requested tab order is not in the hashtable.

### **getFunctionCodeWithId(int)**

This method returns the function code with the requested identifier.

```
public FunctionCode getFunctionCodeWithId(int id)
```

#### **Parameters:**

**id** Integer value of requested function code.

**Returns:** FunctionCode in the hashtable with the requested tab order, or null if the function code with the requested id is not in the hashtable.

### **getPreviousFunctionCode(FunctionCode)**

Given a function code, this method returns the function code with the next lowest tab order.

```
public FunctionCode getPreviousFunctionCode(FunctionCode code)
```

#### **Parameters:**

**code** Returns the function code with the next lower tab order than this FunctionCode.

**Returns:** FunctionCode in the hashtable with the next lowest tab order.

### **getNextFunctionCode(FunctionCode)**

Given a function code, this method returns the function code with the next highest tab order.

```
public FunctionCode getNextFunctionCode(FunctionCode code)
```

#### **Parameters:**

**code** Return the function code with the next highest tab order than this FunctionCode.

**Returns:** FunctionCode in the hashtable with the next highest tab order.

### **getCurrentFunctionCode()**

This method returns the FunctionCode of the active InputField.

```
public synchronized FunctionCode getCurrentFunctionCode()
```

**Returns:** FunctionCode of the active InputField.

### **getTabOrderOfCurrent()**

This method returns the tab order of the FunctionCode of the active InputField.

```
public synchronized int getTabOrderOfCurrent()
```

**Returns:** Tab order of the FunctionCode of the active InputField.

### **getStateId()**

This method returns identifier of the state containing the FunctionCodeList.

```
public int getStateId()
```

**Returns:** State identifier containing the FunctionCodeList.

### **getLowestTabOrder()**

This method returns the lowest tab order of all of the FunctionCodes in the list.

```
public int getLowestTabOrder()
```

**Returns:** Lowest tab order of all function codes in the hashtable.



**clearInputFields(boolean)**

This method clears the input fields of all FunctionCodes in the list.

```
public void clearInputFields(boolean fireEvents)
```

**Parameters:****fireEvents**

True to fire FieldDataUpdatedEvents.

---

**Class LabelInputEvent**

This class is fired when the IOInputQueue receives label input.

**LabelInputEvent Methods**

LabelInputEvent uses the following methods:

**getData()**

Queries the data

**getFunctionCode()**

Queries the function code

**isScannerInput()**

Queries whether the input is scanner input.

**getData()**

Query the data.

```
public String getData()
```

**Returns:** Data

**getFunctionCode()**

Query the function code.

```
public int getFunctionCode()
```

**Returns:** Function code

**isScannerInput()**

Query if this is scanner input. If it is not input from the scanner, it is a keyed label.

```
public boolean isScannerInput()
```

**Returns:** True if this label is scanner input, false if its a keyed label.

---

## **Class QueueLockedException**

This exception is thrown by the `IOPInputQueue` when an attempt is made to add data or a function code to the `IOPInputQueue` when it is locked.

---

## **Class StateForbidsInputException**

This exception is generated by the `IOPInputQueue` when an attempt is made to add data or a function code to the `IOPInputQueue` when the Java I/O processor is in a state that does not allow keyboard input.

---

## Class StateTableProcessor

The state table processor is the state transition engine for the I/O processor. Using the input state table information, it performs state transitions.

### StateTableProcessor Methods

StateTableProcessor uses the following methods:

**activateField(FunctionCode, boolean)**

Sets the currently active input field of the current full-screen state.

**addFullScreenFieldListener(FullScreenFieldListener)**

Adds a full-screen field listener.

**addFullScreenFieldListener(FullScreenFieldListener, int)**

Adds a full-screen field listener that listens for full-screen field events related to a specific state.

**addInputFieldListener(InputFieldListener)**

Adds an input field listener.

**addStateChangeListener(StateChangeListener)**

Adds a state change listener.

**getActiveFunctionCode()**

Returns the FunctionCode object corresponding to the current active input field, if a full-screen state table is loaded and if the current state is a full-screen state.

**getCurrentState()**

Query the current state.

**getCurrentStateId()**

Get the current state identifier.

**getFullScreenAttributes()**

Returns the object containing the full-screen attributes and video parameters for each of the full-screen input areas.

**getGlobals()**

Returns the global parameters.

**getNumberOfStates()**

Gets the maximum number of states in the state table.

**getState(int)**

Gets a POS state.

**isFullScreenTableLoaded()**

Indicates whether an enhanced full-screen state table has been loaded.

**removeFullScreenFieldListener(FullScreenFieldListener)**

Removes a full-screen field listener.

**removeFullScreenFieldListener(FullScreenFieldListener, int)**

Removes a full-screen field listener that is listening for events for a specific state.

**removeInputFieldListener(InputFieldListener)**

Removes an input field listener.

**removeStateChangeListener(StateChangeListener)**

Removes a state change listener.

### **activateField(FunctionCode, boolean)**

Sets the currently active input field of the current full-screen state. The active input field is the field that receives all keyboard and scanner input. Each input field is associated with a FunctionCode as defined in the full-screen state table. When a field is activated, any previously active field in the state is deactivated.

```
public synchronized void activateField(FunctionCode newFCode, boolean fireEvents)
```

#### **Parameters:**

##### **newFCode**

The function code object of the new field to activate.

##### **fireEvents**

Indicates whether field activation and deactivation events should be fired.

### **addFullScreenFieldListener (FullScreenFieldListener)**

Adds a FullScreenFieldListener. All FullScreenFieldListeners are notified when any full-screen input field is activated, deactivated, or updated.

```
public synchronized void addFullScreenFieldListener(FullScreenFieldListener fsfl)
```

#### **Parameters:**

**fsfl** Full-screen field listener.

### **addFullScreenFieldListener (FullScreenFieldListener, int)**

Adds a FullScreenFieldListener that listens for full-screen field events related to a specific state. All FullScreenFieldListeners are notified when a full-screen input field related to the filter state is activated, deactivated, or updated. Because a full-screen state is typically implemented as a single screen, the Java GUI might use this method to listen to field events for a particular state corresponding to a particular screen.

```
public synchronized void addFullScreenFieldListener  
(FullScreenFieldListener fsfl, int filterState)
```

#### **Parameters:**

**fsfl** Full-screen field listener.

##### **filterState**

ID of state listener is interested in.

### **addInputFieldListener (InputFieldListener)**

Adds an InputFieldListener. All InputFieldListeners are notified when any input field is updated.

```
public synchronized void addInputFieldListener(InputFieldListener ifl)
```

#### **Parameters:**

**ifl** Input field listener.

### **addStateChangeListener (StateChangeListener)**

Adds a state change listener.

```
public synchronized void addStateChangeListener(StateChangeListener scl)
```

#### **Parameters:**

**scl** State change listener

### **getActiveFunctionCode()**

If a full-screen state table is loaded, and if the current state is a full-screen state, returns the FunctionCode object corresponding to the current active input field.

```
public FunctionCode getActiveFunctionCode()
```

**Returns:** The FunctionCode of the active InputField, or null if no full-screen field is active.

### **getCurrentState()**

Query the current state

```
public State getCurrentState()
```

**Returns:** Current state

### **getCurrentStateId()**

Get the current state identifier.

```
public int getCurrentStateId()
```

**Returns:** Current state ID

### **getFullScreenAttributes()**

Returns the object containing the full-screen attributes and video parameters for each of the full-screen input areas (prompt area, display as keyed area, and error area). These attributes are defined in the “Common Information” portion of an enhanced full-screen state table. This method should only be called if an enhanced full-screen state table has been loaded.

```
public FullScreenAttributes getFullScreenAttributes()
```

**Returns:** The FullScreenAttribute object populated from the “Common Information” portion of an enhanced full-screen state table.

### **getGlobals()**

Returns the global parameters

```
public Globals getGlobals()
```

**Returns:** Global parameter information

### **getNumberOfStates()**

Get the maximum number of states in the state table.

```
public int getNumberOfStates()
```

**Returns:** Total number of states defined in the input state table.

### **getState(int)**

Get a POS State. The ID is 1-based.

```
public State getState(int id)
```

#### **Parameters:**

**id**        1-based State number

**Returns:** State

### **isFullScreenTableLoaded()**

Indicates whether an enhanced full-screen state table has been loaded.

```
public boolean isFullScreenTableLoaded()
```

### **removeFullScreenFieldListener (FullScreenFieldListener)**

Removes a FullScreenFieldListener.

```
public synchronized void removeFullScreenFieldListener(FullScreenFieldListener fsfl)
```

#### **Parameters:**

**fsfl**        Full-screen field listener.

### **removeFullScreenFieldListener (FullScreenFieldListener, int)**

Removes a FullScreenFieldListener that is listening for events for a specific state.

```
public synchronized void removeFullScreenFieldListener  
(FullScreenFieldListener fsfl,int filterState)
```

#### ***Parameters:***

**fsfl** Full-screen field listener.

#### **filterState**

ID of state listener is interested in.

### **removeInputFieldListener (InputFieldListener)**

Removes a FullScreenFieldListener.

```
public synchronized void removeInputFieldListener(FullScreenFieldListener ifl)
```

#### ***Parameters:***

**ifl** Input field listener.

### **removeStateChangeListener (StateChangeListener)**

Removes a state change listener.

```
public synchronized void removeStateChangeListener(StateChangeListener scl)
```

#### ***Parameters:***

**scl** State change listener

---

## Interface StateChangeListener

This class is the listener interface for receiving state change events. The Java I/O processor changes state due to application input and input state table information.

### StateChangeListener Method

***AWT of Swing components should not be modified without utilizing `SwingUtilities.invokeLater()` to cause a context switch from the `JIOP` thread to the `AWT` thread.***

StateChangeListener uses the following method:

#### **stateChanged(StateChangeEvent)**

Called when the I/O processor state changes.

#### **stateChanged(StateChangeEvent)**

Called when the I/O processor state changes.

```
public abstract void stateChanged(StateChangeEvent sc)
```

#### ***Parameters:***

**sc**      State change event

---

## Class StateChangeEvent

This event indicates that a state change has occurred within the I/O processor.

### StateChangeEvent Methods

StateChangeEvent uses the following methods:

#### **getState()**

Returns the state

#### **getStateId()**

Returns the state identifier

#### **isChangeToCurrent()**

Query if this state change is merely a change to the current state

#### **getState()**

Returns the state.

```
public State getState()
```

**Returns:** New state

#### **getStateId()**

Returns the state identifier.

```
public int getStateId()
```

**Returns:** State identifier

#### **isChangeToCurrent()**

Query whether this state change is merely a change to the current state.

```
public boolean isChangeToCurrent()
```

**Returns:** True if this state change is a change to CURRENT.



---

## Class State

Represents a state as defined by the 4690 input state table. The Java I/O processor is always in some specific state.

### State Methods

State uses the following methods:

**clearFullScreenInputFields(boolean)**

For a full-screen state, this method clears all input fields.

**getActiveFieldFunctionCode()**

If the state is a full-screen state, returns the function code of the currently active field in the state, or null if no field is active.

**getAutoEOFCount()**

Query the data count to trigger auto-EOF

**getAutoEOFFunctionCode()**

Query the function code to be applied for auto-EOF

**getID()**

Returns the integer ID representing the state.

**getLowestTabOrder()**

If the state is a full-screen state, this method returns the lowest tab order of any function code in the state.

**getNumberOfFunctionCodes()**

Returns the number of function codes for this state.

**getPrompt()**

Query for the prompt

**getValidFullScreenFunctionCodes()**

For a full-screen state, this method returns a list of the function codes that are defined for the state.

**getValidFunctionCodes()**

Get the valid function codes from this state

**isAutoEOF()**

Query if this state is defined as automatic end-of-field

**isDataDisplayedPerFCode()**

Indicates if the state has been defined to display the input as defined by each individual function code.

**isDataDisplayedWithEditing()**

Indicates if the state has been defined to display the input with editing.

**isDataDisplayedWithoutEditing()**

Indicates if the state has been defined to display the input with no editing.

**isFullScreenState()**

Indicates whether the state is a full-screen state.

**isKeyboardAllowed()**

Query if this state allows the use of the keyboard

**isScannerAllowed()**

Query if this state allows the use of the scanner

**isSequenceStart()**

Query if this state starts an input sequence

**clearFullScreenInputFields(boolean)**

For a full-screen state, this method clears all input fields.

```
public void clearFullScreenInputFields(boolean generateEvents)
```

**Parameters:****generateEvents**

True if FieldDataUpdatedEvents should be generated.

**getActiveFieldFunctionCode()**

If the state is a full-screen state, returns the function code of the currently active field in the state, or null if no field is active.

```
public FunctionCode getActiveFieldFunctionCode()
```

**Returns:** The function code of the active input field.

**getAutoEOFCount()**

Query the data count to trigger auto-EOF.

```
public int getAutoEOFCount()
```

**Returns:** Auto-EOF count

**getAutoEOFFunctionCode()**

Query the function code to be applied for auto-EOF.

```
public int getAutoEOFFunctionCode()
```

**Returns:** auto-EOF function code

**getID()**

Returns the integer ID representing the state.

```
public int getID()
```

**Returns:** The integer ID of the state.

**getLowestTabOrder()**

If the state is a full-screen state, this method returns the lowest tab order of any function code in the state.

```
public byte getLowestTabOrder()
```

**Returns:** The lowest tab order of any function code in the state.

**getNumberOfFunctionCodes()**

Returns the number of function codes for this state

```
public int getNumberOfFunctionCodes()
```

**Returns:** Number of function codes

**getPrompt()**

Query for the prompt. A prompt can not exist, in which case this method returns null.

```
public String getPrompt()
```

**Returns:** Prompt or null if prompt not set in input state table

### **getValidFullScreenFunctionCodes()**

For a full-screen state, this method returns a list of the functions that are defined for the state. Because the function code list is persistent, this method should be used to obtain full-screen `FunctionCode` objects from the state, rather than the `getValidFunctionCodes()` method.

```
public FunctionCodeList getValidFullScreenFunctionCodes()
```

**Returns:** A list of the valid full-screen function codes defined for the state.

### **getValidFunctionCodes()**

Get the valid function codes from this state

```
public FunctionCode[] getValidFunctionCodes()
```

**Returns:** Valid function codes

### **isAutoEOF()**

Query if this state is defined as automatic end-of-field

```
public boolean isAutoEOF()
```

**Returns:** True if auto-EOF, false otherwise.

### **isDataDisplayedPerFCode()**

Indicates if the state has been defined to display the input as defined by each individual function code.

```
public boolean isDataDisplayedPerFCode()
```

**Returns:** True, if state is defined to display input as defined by each individual function code.

### **isDataDisplayedWithEditing()**

Indicates if the state has been defined to display the input with editing.

```
public boolean isDataDisplayedWithEditing()
```

**Returns:** True, if state is defined to display input with editing.

### **isDataDisplayedWithoutEditing()**

Indicates if the state has been defined to display the input with no editing.

```
public boolean isDataDisplayedWithoutEditing()
```

**Returns:** True, if state is defined to display input without editing.

### **isFullScreenState()**

Indicates whether the state is a full-screen state.

```
public boolean isFullScreenState()
```

**Returns:** True, if state is a full-screen state.

### **isKeyboardAllowed()**

Query if this state allows the use of the keyboard.

```
public boolean isKeyboardAllowed()
```

**Returns:** True, if keyboard input is allowed

### **isScannerAllowed()**

Query if this state allows the use of the scanner.

```
public boolean isScannerAllowed()
```

**Returns:** True, if scanner input is allowed.

### **isSequenceStart()**

Query if this state starts an input sequence.

```
public boolean isSequenceStart()
```

**Returns:** True, if this state starts an input sequence, otherwise, false.

---

## Class FunctionCode

Represents a function key to the application. A function key can have data attributes associated with it (max length, value) and can be special types of functions (motor, clear).

### FunctionCode Methods

FunctionCode uses the following methods:

#### **asString()**

Returns the FunctionCode in a String representation. Used for debugging.

#### **belongsToFullScreenState()**

Indicates whether the function code belongs to a full-screen state.

#### **clearInputFieldData(boolean, int)**

Clears any data and function code in the input field that is associated with this function code.

#### **getColumn()**

Returns starting column position for this function code's field. This is only defined for full-screen data entry function codes that are inside of full-screen states.

#### **getCurrentFieldVideoAttributes()**

Returns attributes that are defined for the input field when it is the current field. See help in the input state table configuration screen. This is only defined for full-screen data entry function codes that are inside of full-screen states.

#### **getDataLengthMaximum()**

Returns the maximum length for the data using information in the input state table.

#### **getDataLengthMinimum()**

Returns the minimum length for the data using information in the input state table.

#### **getDataValueMaximum()**

Returns the maximum value for the data using information in the input state table.

#### **getDataValueMinimum()**

Returns the minimum value for the data using information in the input state table.

#### **getEmptyFieldVideoAttributes()**

Returns attributes that are defined for the field when it is not the current field and the field has no data. See help in the input state table configuration screen. This is only defined for full-screen data entry function codes that are inside of full-screen states.

#### **getHighRange()**

If this function code defines a range of values, gets the high end value.

#### **getId()**

Returns this function code's identifier.

#### **getInputField()**

Returns the InputField instance that is associated with this function code. Used for full-screen function codes that are defined in full-screen states.

#### **getLowRange()**

If this function code defines a range of values, gets the low end value.

#### **getNonEmptyFieldVideoAttributes()**

Returns attributes that are defined for the field when it is not the current field and the field has data. See help in the input state table configuration screen. This is only defined for full-screen data entry function codes that are inside of full-screen states.

**getRow()**

Returns starting row position for this function code's field. This is only defined for full-screen data entry function codes that are inside of full-screen states.

**getStateID()**

Returns the ID of the state that this function code belongs to, or zero if the function code is a common function code.

**getTabOrder()**

Returns the 1-based tab order for this function code's field. This is only defined for full-screen data entry function codes that are inside of full-screen states.

**getTabOrderOfNext()**

Returns the 1-based tab order for the next function codes' field. This is only defined for full-screen data entry function codes that are inside of full-screen states.

**getTabOrderOfPrevious()**

Returns the 1-based tab order for the previous function codes' field. This is only defined for full-screen data entry function codes that are inside of full-screen states.

**hasDataLengthSet()**

Check if data length is available in the table.

**hasDataValueSet()**

Check if data value range is set in the table.

**isAlphanumeric()**

Returns true if alphanumeric data is allowed. This is only defined for full-screen data entry function codes that are inside of full-screen states.

**isAutoTab()**

Returns true if automatic tabbing is enabled for this function code. This is only defined for full-screen data entry functions codes inside of full-screen states.

**isClearKey()**

Is this function code a clear key.

**isDataAllowed()**

Is data allowed for this function code.

**isDataPrecedesFunctionCode()**

Returns true if data should precede this function code.

**isDataRequired()**

Is data required for this function code.

**isFullScreenDataEntry()**

Indicates whether the function code is defined as a full-screen data entry function code within a full-screen state.

**isKeyedLabelMayPrecede()**

Returns true if a keyed label can precede this function code.

**isKeyRange()**

Is this function code a range of values.

**isLeftJustified()**

Returns true, if the data is displayed left justified, or false, if the data is displayed right justified. This is only defined for full-screen data entry function codes inside of full-screen states.

**isManagersKeyRequired()**

Query if a manager's key is required.

**isMotorKey()**

If the function key is a motor key, then pressing the motor key sends the input sequence to the application.

**isUpperCase()**

Returns true if uppercase conversion is enabled for this function code. This is only defined for full-screen data entry functions codes that are inside of full-screen states.

**asString()**

Returns the FunctionCode in a String representation. Useful for debugging.

```
public String asString()
```

**Returns:** Returns the FunctionCode object in a String representation.

**belongsToFullScreenState()**

Indicates whether the function code belongs to a full-screen state.

```
public boolean belongsToFullScreenState()
```

**Returns:** True, if function code belongs to a full-screen state.

**clearInputFieldData(boolean, int)**

Clears any data and function code that is in the input field associated with this function code.

```
public void clearInputFieldData(boolean fireEvents, int source)
```

**Parameters:****fireEvents**

True if FieldDataUpdatedEvents should be generated.

**source**

Source of the clear (either IOPInputQueue, CLEARFIELD or IOPInputQueue, CLEARSEQUENCE)

**getColumn()**

Returns the starting column position for this function code's field. This is only defined for full-screen data entry function codes that are inside of full-screen states.

```
public boolean getColumn()
```

**Returns:** Returns starting column position for the input field associated with this function code.

**getCurrentFieldVideoAttributes()**

Returns attributes defined for the input field when it is the current field. See help in the input state table configuration screen. This is only defined for full-screen data entry function codes that are inside of full-screen states.

```
public short getCurrentFieldVideoAttributes()
```

**Returns:** Returns video attributes for the input field when it is the current field.

**getDataLengthMaximum()**

Returns the maximum length for the data using information in the input state table

```
public int getDataLengthMaximum()
```

**Returns:** Maximum data length

**getDataLengthMinimum()**

Returns the minimum length for the data using information in the input state table.

```
public int getDataLengthMinimum()
```

**Returns:** Minimum data length

### **getDataValueMaximum()**

Returns the maximum value for the data using information in the input state table.

```
public int getDataValueMaximum()
```

**Returns:** Maximum data value

### **getDataValueMinimum()**

Returns the minimum value for the data using information in the input state table.

```
public int getDataValueMinimum()
```

**Returns:** Minimum data value

### **getEmptyFieldVideoAttributes()**

Returns attributes defined for the field when it is not the current field, and the field has no data. See help in the input state table configuration screen. This is only defined for full-screen data entry function codes that are inside of full-screen states.

```
public short getEmptyFieldVideoAttributes()
```

**Returns:** Returns video attributes for input fields that are not the current field, and that are empty.

### **getHighRange()**

If this function code defines a range of values, gets the high end value.

```
public int getHighRange()
```

**Returns:** Range value

### **getId()**

Returns this function code's identifier.

```
public int getId()
```

**Returns:** Identifier

### **getInputField()**

Returns the InputField instance associated with this function code. Used for full-screen function codes that are defined in full-screen states.

```
public int getInputField()
```

**Returns:** Returns InputField associated with the function code.

### **getLowRange()**

If this function code defines a range of values, gets the low end value

```
public int getLowRange()
```

**Returns:** Range value

### **getNonEmptyFieldVideoAttributes()**

Returns attributes defined for the field when it is not the current field, and the field has data. See help in the input state table configuration screen. This is only defined for full-screen data entry function codes that are inside of full-screen states.

```
public short getNonEmptyFieldVideoAttributes()
```

**Returns:** Returns video attributes for input fields that are not the current field, and that contain data.



**getRow()**

Returns starting row position for this function code's field. This is only defined for full-screen data entry function codes that are inside of full-screen states.

```
public boolean getRow()
```

**Returns:** Returns starting row position for the input field associated with this function code.

**getStateID()**

Returns the ID of the state that this function code belongs to, or zero if the function code is a common function code.

```
public int getStateID()
```

**Returns:** ID of the state that owns the function code, or zero if the function code is a common function code.

**getTabOrder()**

Returns the 1-based tab order for this function code's field. This is only defined for full-screen data entry function codes that are inside of full-screen states.

```
public int getTabOrder()
```

**Returns:** Returns tab order for this function code.

**getTabOrderOfNext()**

Returns the 1-based tab order for the next function code's field. This is only defined for full-screen data entry function codes that are inside of full-screen states.

```
public int getTabOrderOfNext()
```

**Returns:** Returns tab order for the function code that follows this function code.

**getTabOrderOfPrevious()**

Returns the 1-based tab order for the previous function code's field. This is only defined for full-screen data entry function codes that are inside of full-screen states.

```
public int getTabOrderOfPrevious()
```

**Returns:** Returns tab order for the function code that preceeds this function code.

**hasDataLengthSet()**

Check if data length is available in the table.

```
public boolean hasDataLengthSet()
```

**Returns:** True, if a data length range is set in the table.

**hasDataValueSet()**

Check if data value range is set in the table.

```
public boolean hasDataValueSet()
```

**Returns:** True, if a data value range is set in the table.

**isAlphanumeric()**

Returns true if alphanumeric data is allowed. This is only defined for full-screen data entry function codes inside of full-screen states.

```
public boolean isAlphanumeric()
```

**Returns:** True, if alphanumeric input is allowed; false, if only numeric data is allowed.

### **isAutoTab()**

Returns true if automatic tabbing is enabled for this function code. This is only defined for full-screen data entry function codes that are inside of full-screen states.

```
public boolean isAutoTab()
```

**Returns:** True, if automatic tabbing is enabled.

### **isClearKey()**

Is this function code a clear key

```
public boolean isClearKey()
```

**Returns:** True, if this function code is the clear key.

### **isDataAllowed()**

Is data allowed for this function code.

```
public boolean isDataAllowed()
```

**Returns:** True, if data is allowed.

### **isDataPrecedesFunctionCode()**

Returns true if data should precede this function code.

```
public boolean isDataPrecedesFunctionCode()
```

**Returns:** True, if data should precede this function code.

### **isDataRequired()**

Is data required for this function code.

```
public boolean isDataRequired()
```

**Returns:** True if data is required.

### **isFullScreenDataEntry()**

Indicates whether the function code is defined as a full-screen data entry function code that is within a full-screen state.

```
public boolean isFullScreenDataEntry()
```

**Returns:** True, if the function code is defined as a full-screen function code in a full-screen state.

### **isKeyedLabelMayPrecede()**

Returns true if a keyed label can precede this function code.

```
public boolean isKeyedLabelMayPrecede()
```

**Returns:** True, if a keyed label can precede; otherwise, false.

### **isKeyRange()**

Is this function code a range of values.

```
public boolean isKeyRange()
```

**Returns:** Returns true if this is a range of keys.

### **isLeftJustified()**

Returns true if data is displayed left justified; false if data is displayed right justified. This is only defined for full-screen data entry function codes inside of full-screen states.

```
public boolean isLeftJustified()
```

**Returns:** True, if left justified; false, if right justified.

**isManagersKeyRequired()**

Query if a manager's key is required.

```
public boolean isManagersKeyRequired()
```

**Returns:** True, if a manager's key is required for this function code.

**isMotorKey()**

Is the function key a motor key, pressing a motor key sends the input sequence to the application.

```
public boolean isMotorKey()
```

**Returns:** True, if this function code is a motor key.

**isUpperCase()**

Returns true if uppercase conversion is enabled for this function code. This is only defined for full-screen data entry function codes that are inside of full-screen states.

```
public boolean isUpperCase()
```

**Returns:** True, if uppercase conversion is enabled.

---

## Class Globals

Represents global information defined in the 4690 input state table. These global attributes are valid in each state in the input state table, unless a particular state redefines the attribute.

### Globals Method

Globals uses the following method:

#### **getGlobalFunctionCodes()**

Gets the global function codes.

#### **getGlobalFunctionCodes()**

Gets the global function codes.

```
public FunctionCode[] getGlobalFunctionCodes()
```

**Returns:** Array of function codes

---

## Class InputSequenceFormatter

Formats and validates user input into input sequences expected by the POS application. It also implements `StateChangeListener` and `LabelInputListener`.

### InputSequenceFormatter Methods

`InputSequenceFormatter` uses the following methods:

#### **addInputSequenceClearedListener(InputSequenceClearedListener)**

Adds an `InputSequenceClearedListener`

#### **clearCurrentInputField()**

Clears the data and function code from the current input field.

#### **getCurrentInputField()**

Returns the current input field. If the current state is a full-screen state, the current field is in the field that corresponds to the active function code.

#### **getNumberOfBuckets()**

Returns the number of buckets for an input sequence. This is 10 unless a full-screen state table is loaded, then it is the highest relative position across all the states.

#### **isDataAvailable()**

Returns true if data is available. (A motor key or error needs to be returned to the application.)

#### **removeInputSequenceClearedListener(InputSequenceClearedListener)**

Removes an `InputSequenceClearedListener`.

#### **addInputSequenceClearedListener (InputSequenceClearedListener)**

Adds an `InputSequenceClearedListener`.

```
public synchronized void  
addInputSequenceClearedListener(InputSequenceClearedListener i)
```

#### **Parameters:**

**i**        Sequence cleared listener

#### **clearCurrentInputField()**

Clears the data and function code from the current input field.

```
public void clearCurrentInputField()
```

#### **getCurrentInputField()**

Returns the current input field. If the current state is a full-screen state, the current field is the field that corresponds to the active function code.

```
public InputField getCurrentInputField()
```

**Returns:** `InputField` corresponding to the active function code.

#### **getNumberOf Buckets()**

Returns the number of buckets for an input sequence. This is 10 unless a full-screen state table is loaded, then it is the highest relative position across all the states.

```
public int getNumberOfBuckets()
```

**Returns:** Number of positions in the input sequence.

#### **isDataAvailable()**

Returns true if data is available. (A motor key or error needs to be returned to the application.)

```
public boolean isDataAvailable()
```

**Returns:** True, if data is available.

### **removeInputSequenceClearedListener (InputSequenceClearedListener)**

Removes an InputSequenceClearedListener.

```
public synchronized void  
removeInputSequenceClearedListener(InputSequenceClearedListener i)
```

#### **Parameters:**

**i**        Sequence cleared listener

---

## Class InputSequenceClearedListener

This class is the interface for notification of input sequence clearing. The input sequence can be cleared due to state change, double clear entry, or other reasons.

### InputSequenceClearedListener Method

InputSequenceClearedListener uses the following method:

#### **sequenceCleared(InputSequenceClearedEvent)**

    Informs the listener the input sequence has been cleared.

#### **sequenceCleared (InputSequenceClearedEvent)**

Adds an InputSequenceClearedListener.

```
public abstract void sequenceCleared(InputSequenceClearedEvent isce)
```

#### ***Parameters:***

**isce**    Event

---

## Class InputSequenceClearedEvent

This class is the interface to the event that indicates that the input sequence has been cleared. The input sequence can be cleared due to a state change, double clear entry, or for other reasons.

### InputSequenceClearedEvent Method

InputSequenceClearedEvent uses the following method:

#### **isDoubleClear()**

Query if the input sequence was a result of double clear entry by the operator.

#### **isDoubleClear()**

Query if the input sequence was a result of double clear entry by the operator.

```
public boolean isDoubleClear()
```

**Returns:** True, if operator entered double clear.

---

## Classes in package com.ibm.OS4690.jiop.util

Java I/O processor classes in package com.ibm.OS4690.jiop.util include:

- SystemMonitor
- ExceptionListener
- ExceptionEvent



## Class SystemMonitor

This class provides a logging and event interface to uncaught exceptions. Uncaught exceptions are logged to the Trace file and to standard err. The Java thread to be monitored should be constructed such that it is in the SystemMonitor ThreadGroup. Subsequently, that thread and other threads in the SystemMonitor group are monitored for uncaught exceptions.

### SystemMonitor Constructor

SystemMonitor uses the following constructor:

#### SystemMonitor()

SystemMonitor constructor

**SystemMonitor():** SystemMonitor constructor

```
public SystemMonitor()
```

### SystemMonitor Methods

SystemMonitor uses the following methods:

#### addExceptionListener(ExceptionListener)

Adds an uncaught exception listener.

#### getInstance()

Gets the system monitor.

#### removeExceptionListener(ExceptionListener)

Removes an uncaught exception listener.

#### uncaughtException(Thread, Throwable)

Logs uncaught exceptions and raises an event.

**addExceptionListener(ExceptionListener):** Adds an uncaught exception listener.

```
public synchronized void addExceptionListener(ExceptionListener uc)
```

*Parameters:*

**uc**      Listener

**getInstance():** Gets the system monitor.

```
public static SystemMonitor getInstance()
```

*Returns:* This method returns the system monitor instance.

**removeExceptionListener(ExceptionListener):** Removes an uncaught exception listener.

```
public synchronized void removeExceptionListener(ExceptionListener uc)
```

*Parameters:*

**uc**      Listener

**uncaughtException(Thread, Throwable):** Logs uncaught exceptions and raises an event.

```
public void uncaughtException(Thread t,  
                               Throwable e)
```

*Parameters:*

**t**      Thread

**e**      Throwable

*Overrides:* uncaughtException in class ThreadGroup

## Class ExceptionListener

This class is the listener interface for receiving uncaught exception events.

### ExceptionListener Method

ExceptionListener uses the following method:

#### **exception(ExceptionEvent)**

Called when the VM detects an exception.

**exception(ExceptionEvent):** Called when the VM detects an exception.

```
public abstract void exception(ExceptionEvent e)
```

*Parameters:*

**e**      Exception event

## Class ExceptionEvent

This class is the event that indicates an uncaught exception has occurred.

### ExceptionEvent Constructors

ExceptionEvent uses the following constructors:

#### **ExceptionEvent(Object)**

Constructs an ExceptionEvent with event source.

#### **ExceptionEvent(Object, Throwable)**

Constructs an ExceptionEvent with event source.

**ExceptionEvent(Object):** Constructs an ExceptionEvent with event source.

```
public ExceptionEvent(Object source)
```

*Parameters:*

#### **source**

Source of the event

**ExceptionEvent(Object, Throwable):** Constructs an ExceptionEvent with event source.

```
public ExceptionEvent(Object source,  
                      Throwable t)
```

*Parameters:*

#### **source**

Source of the event

#### **t**

Throwable

### ExceptionEvent Method

ExceptionEvent uses the following method:

#### **getThrowable()**

Returns the exception that caused the event.

**getThrowable():** Returns the exception which caused the event.

```
public Throwable getThrowable()
```

*Returns:* Thrown exception, null if exception not available.

---

## Redirecting Standard Output from the Terminal

The 4690 Optionals contain these two files that can be used to redirect standard output from the terminal:

- REDIRECT.JAV
- REDIRECT.PRO

The two Java methods used to redirect the output are `System.setOut` and `System.setErr`. The following example shows the code in these files that is used to redirect output. The maximum length of the file name must be 25 characters.

```
.
.
.
{
if (stdoutFileName !=null)
System.setOut (new PrintStream (new FileOutputStream (stdoutFileName)))

if (stderrFileName !=null)
System.setErr (new PrintStream (new FileOutputStream (stderrFileName))).
.
.
.
}
```

`stdoutFileName` and `stderrFileName` must be replaced with `R::` followed by a unique name up to a maximum of 25 characters, including the `R::` characters. For example, this could be a replacement for `stdoutFileName`:

```
R::ADX_SPGM:stdout.Out
```

---

## Using an ANPOS Keyboard with Java

An ANPOS keyboard combines alphanumeric PC-style keys with POS-specific keys.

In order to allow alpha characters to be accepted by the JIOP, the current state must be defined such that the DATA DISPLAY option is set to 4, and the alphanumeric data field within the function code definition must be set to Y. If you want to always allow alpha characters regardless of the current state, a JIOP property can be set to always allow alpha. The following JIOP property must be defined within J.PRO and set to any value, `JIOP.alwaysAllowAlpha=xxx`, where `xxx` can be any characters and any length that you choose. To change back to the standard JIOP behavior, the JIOP property, `JIOP.alwaysAllowAlpha=xxx`, must be removed from J.PRO. See “Capturing a Java IO Process Trace” on page 654 for more details about running with the J.PRO properties file.

There are two ways that a Java application can receive input from an ANPOS keyboard:

- The Java application can receive keystrokes through the AWT event queue. This is the normal mechanism for Java programs to receive keyboard input. GUI components are able to directly process this type of keyboard input. See the JDK documentation for more information on processing AWT keyboard input. The limitation to this method is that the POS-specific keys are ignored.
- The Java application can instantiate an instance of the `JavaPOS` class `POSKeyboard`. All of the keys on the ANPOS keyboard are delivered as `DataEvents` to the `POSKeyboard.dataOccurred()` method. While all keys on the ANPOS keyboard are recognized, GUI components cannot directly recognize this input because it is not sent to the AWT event queue. For example, an operator cannot use the ANPOS keyboard to open a browser, navigate through the keyboard, and fill in a Web page form when the ANPOS keyboard input is handled by `JavaPOS`.

**Note:** When Java IO Processor redirection is configured, the Java IO Processor is loaded in place of the standard IO Processor. The Java IO Processor uses the `JavaPOS` `POSKeyboard` class to receive keyboard input. Therefore, ANPOS keyboards (when used with Java GUIs and IO Processor

redirection) are in JavaPOS mode by default. If the Java GUI must display a browser window, then it must be able to switch the keyboard to AWT mode while the browser window is active.

The following section documents considerations when using an ANPOS keyboard in different configurations with Java applications.

## Port 5-Attached ANPOS Keyboard

An ANPOS keyboard that is attached through port 5 of a 4694 register is always in JavaPOS mode. There is no way to receive AWT keyboard events for this configuration.

## PS/2-Attached ANPOS Keyboard

An ANPOS keyboard that is attached through the standard PS/2 keyboard port can be switched between AWT and JavaPOS mode under software control. The keyboard must be configured as a shared ANPOS keyboard in the device group. See the `TerminalApplicationServices.setTermJavaKeysToJavaPOS()` and `TerminalApplicationServices.setTermJavaKeysToAwt()` methods.

## USB-Attached ANPOS Keyboard

- | An ANPOS keyboard that is attached through a USB port on a SurePOS 300/700 or TCxWave 6140
- | Series register can be switched between AWT and JavaPOS under software control. This keyboard must
- | be configured as a shared ANPOS keyboard. See the
- | `TerminalApplicationServices.setTermJavaKeysToJavaPOS()` and
- | `TerminalApplicationServices.setTermJavaKeysToAwt()` methods “`setTermJavaKeysToAwt()`” on page 529.

---

## Using an ANPOS Keyboard with Java IO Processor

The Java IO Processor handles right and left arrow keys, forward and back tab keys, backspace, delete, and insert keys if they are defined correctly in the keyboard layout.

When using an ANPOS keyboard with the Java IO Processor, use the following table to define the function codes for these keys.

*Table 53. Defining function codes for keys when using an ANPOS keyboard with the Java IO Processor*

Key	Function Code
forward tab	16
back tab	17
delete	18
backspace	23
right arrow	26
left arrow	27
insert	29

---

## Capturing a Java IO Process Trace

The Java IO Processor includes a JIOP trace log that can be enabled for problem determination. To capture the JIOP trace information, the Java class must be configured to run with a properties file. You must perform these steps:

1. Create a file named `J.PRO` on the controller in the root directory of the C: drive, so that all terminals can write JIOP trace entries into the trace file. Add the following line to this property file, where *filename* is the name of the JIOP trace file to be written:

```
JIOP.debug=filename
```

The *filename* should be qualified with R:: and can be no more than 25 characters including the R:: characters.

2. When configuring the terminal load definition, add the following line before the Java class to be run, where *YourClassName* is the name of the Java class to execute in the terminal:

```
-Dprops=r::c:\j.pro YourClassName
```

There are four ways to collect the JIOP trace information without dumping the terminal:

- All terminals write JIOP trace entries into a single JIOP trace file.

To use this method, specify a file name on the *JIOP.debug* property.

```
JIOP.debug=r::c:\trace.out
```

- Each terminal writes JIOP trace entries into an individual JIOP trace file.

To use this method, set the *JIOP.debug* property to **byterminal** instead of to an actual file name.

The JIOP trace file is written to the root directory (the C: drive) of the controller for each register. The file name is Jxxx.OUT, where xxx is the three-digit terminal number.

```
JIOP.debug=byterminal
```

- Each terminal writes JIOP trace entries into a sequentially, numbered JIOP trace file.

To use this method, set the *JIOP.debug* property to **byterminal:n** instead of to an actual file name, where **n** is the number of trace files to be written before overwriting the previous JIOP trace files for each register.

The JIOP trace files are written to the root directory (the C: drive) of the controller for each register. The file name is Jxxx.yyy where xxx is the three-digit terminal number and yyy is a sequential number beginning with 000 and going to n-1.

```
JIOP.debug=byterminal:10
```

- Each terminal writes JIOP trace entries into a sequentially, numbered JIOP trace file. Each file has a maximum number of trace entries.

To use this method, set the *JIOP.debug* property to **byterminal:n:m** instead of to an actual file name, where **n** is the number of trace files to be written before overwriting the previous JIOP trace files for each register, and **m** is the maximum number of trace entries in a JIOP trace file. When the trace file contains the specified number of trace entries, a new trace file is started for the terminal.

The JIOP trace files are written to the root directory (the C: drive) of the controller for each register. The file name is Jxxx.yyy where xxx is the three-digit terminal number and yyy is a sequential number beginning with 000 and going to n-1.

```
JIOP.debug=byterminal:10:250
```

If you want to avoid network traffic and disk activity on the controller, there is an in-memory trace that requires dumping the terminal. The in-memory trace sends the JIOP traces to a circular buffer in memory. To use this method, set the *JIOP.debug* property to **inmemory:xxx** instead of to an actual file name, where **xxx** specifies the number of kilobytes in the buffer.

```
JIOP.debug=inmemory:100
```

If you use the in-memory trace, depending on the memory load on the terminal, a value between 100 and 250 is recommended for **xxx**. If you do not specify a property file, the default buffer size is 100 kilobytes.

The JIOP traces stored at the controller are not intended for a store environment, except when needed for problem determination.

---

## Capturing a Trace Dump File

Trace dump files are written when there is a Java exception. At initialization, the JIOP attempts to determine the name of the next available trace dump file.

Trace dump files are written to the ADX\_SDT1 directory. They are named:

`TRACExxx.yyy`

where *xxx* is the terminal number and *yyy* is the next available number from 000 to 999. If all of the available numbers have been used, then the oldest trace file number is used and the previous data in that trace file is overwritten.

You can also specify the maximum number of trace files by setting the property:

`JIOP.maxTraceFiles=n`

in a properties file, where *n* is the maximum number of trace files to write per register before starting over at 000.

Lastly, when configuring the terminal load definition, add the following line before the Java class to run, where *YourClassName* is the name of the Java class to execute in the terminal:

`-Dprops=r::c:\j.pro YourClassName`

---

## Chapter 21. DBCS support for Java 2

---

### Sample DBCS program

There is no special procedure to compose a DBCS GUI application except that the content to be displayed contains some DBCS characters. In addition, the DBCS font files are only installed for DBCS language builds; the non-DBCS language builds do not have these fonts installed.

The following sample illustrates how to compose a DBCS GUI application:

```
import java.awt.*;
import java.awt.event.*;

public class DBCSGUITest {

    public static void main( String[] args ) {
        Frame frame = new Frame("DBCS GUI Test");
        frame.setFont(new Font("Serif", Font.PLAIN, 16));
        DBCSGUITest app = new DBCSGUITest();
        frame.add(app.init());
        frame.addWindowListener( new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
        frame.pack();
        frame.show();
    }

    public FontPanel init() {
        FontPanel panel = new FontPanel();
        return panel;
    }

    class FontPanel extends Panel implements ItemListener {
        private Label label = new Label("DBCS GUI TEST---\u4e2d\u6587");
        private List familyList = new List();
        private List styleList = new List();
        private List sizeList = new List();

        public FontPanel() {
            setLayout(new BorderLayout());
            Panel fontPanel = new Panel();
            fontPanel.setLayout(new GridLayout(1,3));
            String fontNames[] = getToolkit().getFontList();
            for(int i=0; i < fontNames.length; ++i) {
                familyList.add(fontNames[i]);
            }
            familyList.addItemListener(this);
            familyList.select(0);
            fontPanel.add(familyList);

            styleList.add("Plain");
            styleList.add("Bold");
            styleList.add("Italic");
            styleList.add("BoldItalic");
            styleList.addItemListener(this);
            styleList.select(0);
            fontPanel.add(styleList);

            String sizes[] = {"12", "14", "16", "18", "24", "36"};
            for(int i=0; i < sizes.length; ++i) {
                sizeList.add(sizes[i]);
            }
            sizeList.addItemListener(this);
            sizeList.select(0);
        }
    }
}
```

```

        fontPanel.add(sizeList);
        add(BorderLayout.NORTH, fontPanel);
        add(BorderLayout.CENTER, label);
    }

    public void itemStateChanged(ItemEvent event) {
        Font font = new Font(familyList.getSelectedItem(),
                            styleList.getSelectedIndex(),
                            Integer.parseInt(sizeList.getSelectedItem()));
        label.setFont(font);
        invalidate();
    }

} // END class FontPanel

} // END class DBCSGUITest

```

For an example of a program on a Microsoft Windows system, see Figure 19. On 4690, the output is similar when using Java 2 with the DBCS fonts installed.



Figure 19. Output Example of DBCS Program

## Input Method Operation

Input methods (IME) are software components that allow text to be entered by using the keys on the keyboard differently than how the keys are normally used. IMEs are commonly used to enter Japanese, Chinese, or Korean languages, which use thousands of different characters, on keyboards that have far fewer keys. In Java 2, four types of IMEs for DBCS languages are enabled in 4690 OS Version 3 and later versions.

**Note:** The IME jar files and other related files for a particular language are shipped only with the OS4690 build for that particular language. No IMEs are shipped with the non-DBCS builds.

**Note:** The IME extensions are not supported in Java 6.

## Select and Activate an Input Method (IME)

To activate an IME, from the Java GUI menu, choose **Select Input Method**. A menu appears containing a full list of installed IMEs. Select the specific IME from this menu (see Table 54).

Table 54. IMEs and File Names

Input Method (IME)	File Name
Simplified Chinese IME <ul style="list-style-type: none"> <li>• Pinyin Input Method</li> <li>• ABC Input Method</li> </ul>	m:/java2/jre/lib/ext/SChineseIM.jar



Table 54. IMEs and File Names (continued)

Traditional Chinese IME • Phonetic Input Method • Tsangjye Input Method	m:/java2/jre/lib/ext/TChineseIM.jar
Japanese IME	m:/java2/jre/lib/ext/JapaneseIM.jar
Korean IME	m:/java2/jre/lib/ext/KoreanIM.jar

## Common Features of Input Methods (IME)

These are some of the common features of input methods installed in 4690 OS V3R1 and later releases:

- General input process

Typically, when using input methods to type text in Chinese, Japanese, or Korean, a sequence of several characters needs to be typed and then converted in one chunk. Conversion might need to be retried because there could be several possible translations. This process is called *composition*. The text that the input method is working on is called *composed text*. When the final conversion is confirmed and the text is committed the process is complete.

- On-the-spot style input

During composition, the composed text is displayed in the context of the document that it is included in; albeit, in a style that indicates that the text still needs to be converted or confirmed by the input method. This is called on-the-spot style input.

- Status and Look-up windows

Two kinds of windows are used in input methods to communicate with the user. They are:

- Status window

This type of window appears when an input method is activated. The status window provides information about the current state of the input method, such as the selected language and input modes.

- Look-up window

This type of window is shown when the input matches numerous interpretations. The window contains a list of possible interpretations of the input and allows you to choose the correct interpretation.

- Input modes

Input methods control the input processing through input modes, including character mode, character size, and so on. Typically, input modes are combined together to affect the input.

## Simplified Chinese Input Method

The supported character groups for the Simplified Chinese Input Method (SCIM) are ASCII (English) and Simplified Chinese.

### SCIM Features

SCIM features the following characteristics:

- Supports two commonly used input methods:
  - Pin Yin: An IM based on phonetic combinations
  - ABC: An intelligent IM based on phonetic combinations

In order to input a Simplified Chinese character, Pin Yin SCIM requires complete phonetic symbols that represent the character. The ABC SCIM allows simplified phonetic symbols that allow the input of key symbols instead of all symbols. Also, the inputting of words is enabled in ABC SCIM and is disabled in Pin Yin SCIM.

**Note:** The pronunciation of Simplified Chinese is represented by phonetic symbols called Bo-Po-Mo-Fo. There are 25 phonetic symbols. A Chinese character is represented by up to three phonetic symbols.

- Supports two input modes:
  - Character mode: Chinese or English
  - Character size: full-width or half-width

When the character mode is Chinese, only full-width Chinese characters can be input regardless of the setting of character size, but full or half symbol characters can be input depending on the size setting.

When the character mode is English, ASCII characters in both full-width and half-width can be input depending on the size setting.

## How to Input Using SCIM

From the system menu, SCIM is invoked by selecting **Pin Yin Input Method** or **ABC Input Method**. Each phonetic symbol is assigned to a key. Input the phonetic symbols to an on-the-spot, pre-editing, drawing area. Pin Yin and ABC SCIMs generate a list of candidates that appear in a pop-up window. Choose the appropriate character by selecting the candidate number. Invalid input generates a warning beep.

The following special keys for the SCIM are defined on the Simplified Chinese 101-key keyboard (see Table 55).

Table 55. SCIM Special Keys and Functions

Key Function	Key	Description of Function
English/Chinese toggle key	Ctrl-Space	Toggles between English and Chinese input modes
Half/Full-Width toggle key	Shift-Space	Toggles between half-width and full-width input modes
Candidate key	Space / Enter	Displays the candidate list in an auxiliary window, if needed, under Chinese input mode
Conversion key	Enter / number keys	Converts phonetic symbols into characters under Chinese input mode
Non-Conversion Shift key	Enter	Interprets a phonetic symbol as a character under English input mode

## Traditional Chinese Input Method

The supported character groups for the Traditional Chinese Input Method (TCIM) are ASCII (English) and Traditional Chinese.

### TCIM Features

TCIM features the following characteristics:

- Supports two commonly used input methods:
  - Tsangjye: An IM based on the construction of Chinese characters
  - Phonetic: An IM based on inputting a character based on its pronunciation with an enabled legend
 Tsangjye supports radicals to generate a character while Phonetic supports phonetic symbols.

**Note:** The pronunciation of Traditional Chinese is represented by phonetic symbols called Dsu-Yin or Bo-Po-Mo-Fo. There are 37 phonetic symbols and four intonation indicators. Chinese characters are represented by up to three phonetic symbols. Each character can include one intonation symbol. The omission of an intonation symbol implies a fifth intonation accent.

- Supports two input modes:
  - Character mode: Chinese or English
  - Character size: full-width or half-width

When the character mode is Chinese, only full-width Chinese characters can be input regardless of the setting of character size, but full or half symbol characters can be input depending on the size setting.

When the character mode is English, ASCII characters in both full-width and half-width can be input depending on the size setting.

## How to Input Using TCIM

From the system menu, TCIM is invoked by selecting **Tsangjye Input Method** or **Phonetic Input Method**. Each radical or phonetic symbol is assigned to a key. Input the radical or phonetic symbols to an on-the-spot, pre-editing, drawing area. Tsangjye and Phonetic TCIMs generate a list of candidates that appear in a pop-up window. Choose the appropriate character by selecting the candidate number. Invalid input generates a warning beep.

The following special keys for the TCIM are defined on the Traditional Chinese, 106-key, keyboard (see Table 56).

Table 56. TCIM Special Keys and Functions

Key Function	Key	Description of Function
English/Chinese toggle key	Ctrl-Space	Toggles between English and Chinese input modes
Half/Full-Width toggle key	Shift-Space	Toggles between half-width and full-width input modes
Candidate key	Space / Enter	Displays the candidate list in an auxiliary window, if needed, under Chinese input mode
Conversion key	Enter / number keys	Converts phonetic symbols into characters under Chinese input mode
Non-Conversion Shift key	Enter	Interprets a phonetic symbol as a character under English input mode

## Japanese Input Method

The supported character groups for the Japanese Input Method (JIM) are ASCII (English), Katakana, Hiragana, and Kanji.

### JIM Features

JIM is a sophisticated IME that provides Japanese input. JIM features the following characteristics:

- Supports Romaji to Kana character conversion (RKC).

For users that are familiar with alphanumeric keyboards, it is easier to key in the phonetic sounds rather than the Hiragana or Katakana characters. JIM provides Romaji-to-Kana conversion (RKC) allowing the phonetic sounds of Hiragana or Katakana characters to be typed in on an alphanumeric keyboard.

- Supports Kana to Kanji character conversion (KKC).

The KKC technology is based on the fact that every Kanji character or set of Kanji characters has a phonetic sound or sounds that can be expressed by Katakana or Hiragana characters. It is much easier to input Hiragana or Katakana characters than Kanji characters. The JIM analyzes the phonetic values of the Katakana or Hiragana characters to determine the best Kanji-character equivalent. Such phonetic analysis depends on the dictionary and tables provided to the JIM.

- Supports three different input modes.

JIM has three different modes that can be used to control the input processing:

- Keyboard Mapping  
Allows invocation of alphanumeric, Katakana, or Hiragana modes.
- Character Size  
Includes Hankaku (half-width) and Zenkaku (full-width) character input.
- RKC on/off

Inputs Kana directly or invokes the composition to input Kana with a combination of alphabetic characters. The composition facility allows processing of characters before they are committed to the application.

When the keyboard mapping mode is alphanumeric and the character size mode is Hankaku, JIM maps keys to Romaji characters. This mode combination is known as the "English" mode. When in the English mode, the RKC

## How to Input Using JIM

### Set Input Modes

The first step to using JIM is to select the correct input mode. Some function keys are used to set favorite keyboard mapping, character size, and RKC mode.

**Keyboard Mapping:** The three selections available for setting the keyboard mapping are Hiragana, Katakana, and Alphanumeric. The following keys are used to toggle between input modes by the JIM (see Table 57):

Table 57. Keyboard Mapping Keys and Function

Key	Function
Katakana (Shift-Hiragana)	Katakana
Eisu	Alphanumeric
Hiragana	Hiragana

When one of these keys is pressed, keyboard mapping enters the state associated with the key. This state is maintained until one of the other function keys is pressed. The initial keyboard mapping state Hiragana.

**Character Size:** A subset of the Japanese character set is represented in both full-width and half-width. Kanji ideographic characters are usually full-width. The phonetic and ASCII characters have both full-width and half-width representations. The character size is controlled by pressing the Zenkaku\_Henkaku key, which toggles between full-width and half-width.

The following key is used to toggle between input modes by the JIM (see Table 58).

Table 58. Character Size Keys and Function

Key	Function
Zenkaku_Hankaku	Toggles between full-width or half-width modes

**Romaji-to-Kana Conversion (RKC):** The following key is used to toggle among input modes by the JIM (see Table 59).

Table 59. Romaji-to-Kana Toggle Key and Function

Key	Function
Romaji (Alt-Hiragana)	Enables/disables Romaji-to-Kana conversion

**Kana Input:** Kana input is different between RKC mode on and off.

- RKC on

When operating in Romaji-To-Kana conversion mode, you can input Hiragana characters by typing their Romaji phonetic characters. In detail, you produce a Hiragana character by typing 1 to 3 Romaji alphabetic keys that compose the phonetic sound of the Hiragana character. For example, when entering the Kanji characters for the phonetic sound "k-a-n-j-i", you must do the following steps:

1. Set the keyboard mapping to the Hiragana state.

2. Press **Alt-Hiragana** to enable Romaji-to-Kana mapping. This key sequence invokes the alphanumeric keyboard.
  3. Press the keys that spell "kanji". As each phonetic sound is completed, a Hiragana character is displayed.
- RKC off

When you invoke the Hiragana or Katakana with RKC off mode, each key is mapped to a phonetic character within the respective character set. For example, if you press **q**, a Hiragana character pronounced "ta" is produced during Hiragana state, a Katakana character pronounced "ta" is produced during Katakana state, or a Romaji "a" is produced during Eisu state. On Japanese Toshiba keyboards, the tops of the keys show all three symbols.

Some keys have two Hiragana or Katakana characters assigned. For example, the 7 key has large and small Hiragana characters both having the pronunciation "ya". The small characters are used to express special phonetic sounds. These characters can be distinguished by using the shift key. For example, if you press **7**, a large "ya" is generated. Press **Shift+7** and a small "ya" appears.

*Kanji Input:* When keyboard mapping is in Hiragana or Katakana state, Hiragana or Katakana characters within the composed text can be converted into Kanji characters by pressing conversion keys such as Henkan. When the Henkan key is pressed, the most likely candidate associated with the phonetic Hiragana sound is displayed. Pressing this key repeatedly shows other candidates. If the count of candidates is more than 3, a look-up window containing all candidates is shown for you to select what you want.

During the composition process, the composed text is partitioned into segments that can be considered Kanji words. Once a string of kana characters is converted into a candidate, it is treated as one of these convertible segments. While the composed text is displayed, JIM uses the cursor key and other keys to manipulate the string.

To commit the composed text to the program, press **Enter**. In this case, the Enter key code itself is not sent to the program, only the string.

The following keys are also used when composing a Kanji string (see Table 60).

*Table 60. Keys for Composing a Kanji String*

Key	Function
Henken (Convert + Enter)	Converts Kana to Kanji or gets the next candidate
Space	Converts Kana to Kanji or gets the next candidate
Muhenkan (No-convert)	Leaves Kana characters as is
Ctrl + Space	Converts and activates a look-up window
Shift + Space	Converts to the previous candidate
Insert	Restores to the Kana character
Backspace	Restores to the Kana character or deletes the character before the cursor
F6	Converts to Hiragana characters
F7	Converts to full Katakana characters
F8	Converts to half Katakana characters
Up/Down arrow	Moves to previous/next candidate when a look-up window is active
Esc	Discards the current pre-edit string
Enter	Commits
Down arrow	Commits when NO look-up window is active

Table 60. Keys for Composing a Kanji String (continued)

Numeric keys (1–0)	Commits when a look-up window is active
--------------------	---

## Korean Input Method

The supported character groups for the Korean Input Method (KIM) are ASCII (English) and Hangul (Korean characters). The Hangul code set includes Hangul characters only. Hanja (Chinese) characters are not supported.

### KIM Input Method Features

KIM is a simple input method having the following features:

- The look-up window is not used in the KIM input method.

In KIM, because certain input generates a single candidate, the look-up window is unnecessary to use.

- Supports only one input mode.

The only input mode used in KIM is character mode. You select either Hangul or English for this mode. Hangul mode is used to input full-width Hangul and half-width symbol characters. English mode is used to input half-width ASCII characters. The default mode is set to Hangul.

### How to Input Using KIM

From the system menu, KIM is invoked by selecting **Korean Input Method**. Once Hangul mode is invoked, the KIM composes incoming consonants and vowels according to Hangul composition rules. A Hangul character is composed of a consonant followed by a vowel. A final consonant is optional. If incoming characters violate the construct rule, a warning beep is sounded.

To allow for these conversions, the following special key appears on the Korean keyboard (see Table 61).

Table 61. Korean Conversion Key and Function

Key Function	Key	Description of Function
Hangul/English toggle key	Ctrl-Space	Toggles between Hangul and English modes

---

## Chapter 22. Creating 32-Bit Programs Using VisualAge C/C++

This chapter contains information on using VisualAge C/C++ to create 32-bit executable applications and DLLs that can be used with the 4690 Operating System. In addition to allowing you to create stand-alone applications and DLLs, the operating system supports a Java Native Interface (JNI) that is compatible with the VisualAge C/C++ compiler. This allows you to integrate VisualAge C/C++ programming code with Java.

**Note:** Except as documented below, only the basic C and C++ runtime functions are supported in 4690 OS V2 or higher. This includes the C++ new and delete operators, exception handling, and iostream libraries. The OpenClass and complex number libraries are not supported.

This chapter contains only the information that is specific to the 4690 32-bit development and runtime environment. For general C and C++ language, runtime, and compiler information, refer to the online documentation shipped with the VisualAge compiler.

---

### Development Platform

The compiler used to create 32-bit applications for 4690 OS V2 is the VisualAge C/C++ Compiler Version 3.6.5 for Windows NT. This compiler is currently shipped as part of VisualAge C++ Professional version 4.0 and runs on Microsoft Windows NT 4.0. In addition to installing the compiler, it is recommended that you install any fixpacks available for it.

### Files Provided for Development

The 4690 Optionals contain the header files, libraries, and other files needed to develop 32-bit 4690 applications. The files are packaged in VADEVENV.ZIP, which is located on the 4690 Optionals. To use this file, unzip it to a directory on your NT development system. If necessary, you should select the option to create the directories stored in the .ZIP file.

When you unzip VADEVENV.ZIP, a directory called OS4690 is created. Within this directory, there are several directories containing the files needed to compile and link your program. These directories are:

**bin** Binary files (DLLs and executable programs).

**lib** Library and .OBJ files. This directory contains the import and static link libraries used to link 32-bit programs. The names and use of these files are described in "Linking Your Applications" on page 667.

**include**

Directory containing the header files needed to use additional functions provided by the 32-bit development environment (in addition to the C/C++ run-time functions). The header files are divided into several different subdirectories according to their use. To use the header files in these directories, add the appropriate directory name to the *INCLUDE* environment variable or the compiler's */I* option. For example, to use the CAPI function, specify *D:\OS4690\INCLUDE\CAPI*. The following subdirectories are provided:

capi	Contains the header files used to create programs that use the controller and terminal C APIs. See Chapter 17, "Designing Applications with Other Languages," on page 333 and Chapter 19, "Designing Terminal Applications With C Language," on page 397 for more information on using the CAPI functions.
nonfpu	Updates to C run-time header files required to use the 4690 run times with machines that do not contain a floating point coprocessor (386- and 486SX-based systems). See "time_t Changes" on page 674 for more information.
oldtime	Updates to C run-time header files required to use the 4690 run times with existing code that assumes <i>time_t</i> is a long. See "time_t Changes" on page 674 for more information.



tcipip                      Contains the header files used to create applications that use the TCP/IP socket library. See the *4690 OS: Communications Programming Reference* for more information on using TCP/IP.

## Support DLLs Provided by the Operating System

The following DLLs are installed with the operating system and are required for 32-bit applications created with the VisualAge compiler to operate.

File	Description
CAPIC.DLL	Controller CAPI library DLL
CAPIT.DLL	Terminal CAPI library DLL
CRTFMI36.DLL	C runtime DLL (used when the runtimes are not linked in statically)
OSLIB.DLL	OS interface DLL (always required)
TCPIP.DLL	TCP/IP library DLL
CRTFRT36.DLL	C runtime message DLL (used when messages are not directly bound into a .386 file)

## Compiling Your Program

The following flags are required when compiling 32-bit programs for 4690 OS V2 or higher. Refer to the online documentation installed with the VisualAge compiler for more information on these flags.

Flag	Description
/Gm+	Compiles the code with multi-threaded support and allows the program to be properly linked with the multi-threaded C runtime library (which is the only one provided).
/Gn+	Prevents inclusion of default library information, which ensures you don't accidentally link in a Windows NT library. This can also be accomplished by using the /B"/NOD" option in the link stage.
/B"/NOFIXED"	Ensures that fixes are included in the generated executable file. This is necessary because 4690 OS does not necessarily load the program at the same address as Windows NT.
/B"/ALIGNADDR:0x1000"	Reduces the amount of memory usage required to load a program in 4690. There is no benefit to using values less than 0x1000 (4 Kb).

The following flags are optional when compiling 32-bit applications for 4690 OS V2 or higher.

Flag	Description
/Fe	Can be used when building program files to name the resulting executable file. Normally, the file will have the extension .EXE. However, the operating system requires that 32-bit applications have the extension .386.
/Ge[+ -]	Can be used to generate either an executable program or .DLL as needed.
/Gd+	Compile with "/Gd+" when you are using the version of the C run-time library that is dynamically linked (CRTFMI36.LIB/CRTFMO36.LIB). Programs created with this flag are smaller because they use the DLL version of the C run-time library that is shipped with 4690 OS. Any software maintenance applied to this DLL is automatically used by the application. In addition, when multiple 32-bit applications are using the same DLL, the code is shared among all applications, which results in less overall memory usage.



If you want to use *time\_t* on a system without an FPU, use `/ld:\OS4690\INCLUDE\NONFPU`. This ensures that the updated *time\_t* related headers are included before the original C run-time headers. See “time\_t Changes” on page 674 for more information.

---

## Linking Your Applications

Because the compiler often does additional work before linking (such as binding in resources and generating export and .DEF files), it is recommended that you use the *icc.exe* command to link your program. Do not attempt to link directly using *ilink.exe*. When linking the application, the OSLIB.LIB library file is needed. This is the import library for the 4690 OS interface routines.

You must also link the C run-time code. You can link this as either a static or dynamic link. However, dynamic linking is preferred. Using dynamic linking, the C run-time code is read from a .DLL file at run time, which reduces the size of the .386 file and enables the code to be shared among all 32-bit processes on the system. To link with the dynamic version of the C run-time library, you need to compile your code with the `/Gd+` flag and link with the following libraries.

- CRTFMI36.LIB
- CRTFMO36.LIB

CRTFMI36.LIB is the import library for the C run-time DLL, and CRTFMO36.LIB is a static object library that contains modules required to be linked directly into every .EXE or .DLL file.

To link with the static version of the C run-time library, you need to compile your code with the `/Gd-` flag and link with the CRTMFL36.LIB library, which is the static C run-time library.

## Optional Link Libraries

If you want to use functions in the TCP/IP or CAPI libraries, you will also need to link with one or more of the following libraries.

- TCPIP.LIB — Import library for the TCP/IP routines
- CAPIC.LIB — Import library controller CAPI routines
- CAPIT.LIB — Import library terminal CAPI routines

**Note:** You should link with either the controller or terminal library, not both.

## Wildcard Expansion and Multi-byte Argument Support

You receive in your program arguments that are typed on the command line as arguments to the *main()* function. To allow the global characters ? and \* in file and path name arguments at the command prompt, or to allow multi-byte characters, a program must be linked with a replacement parameter handling routine in one of the following OBJ files, which are located in the LIB directory:

- SETARGV.OBJ — Single-byte command line processor with wildcard expansion. The expanded list contains only non-hidden files.
- MSETARG.OBJ — Multi-byte command line processor without wildcard expansion. The expanded list contains only non-hidden, non-system files.
- MSETARGV.OBJ — Multi-byte command line processor with wildcard expansion. The expanded list contains only non-hidden, non-system files.
- MSETARGA.OBJ — Multi-byte command line processor without wildcard expansion. The expanded list contains directory names as well as the names of files with the hidden and/or system attributes set.
- MSTARGVA.OBJ — Multi-byte command line processor with wildcard expansion. The expanded list contains directory names as well as the names of files with the hidden and/or system attributes set.

**Note:** When linking with these object files, you'll also need to use the `/B"/NOE"` compiler option.

---

## Messages

Messages that the C run-time library uses (when you call *assert()* or *perror()*) are not hardcoded into the run-time code. These messages are dynamically loaded at run time from the resources section of a .386 or .DLL file. By default, run-time messages are read from CRTFRT36.DLL. The English language version of this .DLL file is installed automatically with the operating system. No other language versions are currently supplied.

---

## Creating DLLs

Refer to the compiler documentation for information about creating .DLL files. However, when creating definition (.DEF) files and import libraries, do not attempt to import symbols from other .DLL files by ordinal value only. 4690 OS V2 or higher supports importing symbols by name only. Any .DEF files you create should not contain any ordinal values.

**Note:** Some VisualAge tools (such as ILIB) will sometimes put ordinal values in the .DEF files it creates. The ordinal values should be removed before using the .DEF files to create import libraries.

---

## C Run-time Environment Variables on 4690

In 4690, the environment settings are stored in a set of three files located in the ADX\_SDT1 directory. The files consist of a number of name=value pairs, one per line in each file. The environment settings for a given process will be read from up to two of these three files. If either file is missing, it is skipped without reporting any errors.

Before the environment files are read in, the following logical names are queried and placed into the environment. If these values are set from within the environment files, the logical name values will be overridden. If one of these values appears in both the process and system logical name tables, the value in the process table will be used.

**Note:** If environment variables are set for any of the names in the list below, the process logical name table for the current process will be updated with the new value.

- domain
- hostname
- lc\_all
- namesrvr
- path
- shell
- switchar
- tz

In addition, the following environment variables have unique processing done to them:

### CLASSPATH

Before the environment files are processed, the values of the logical names *java2cp1* through *java2cp8* are queried and the C run-time value of CLASSPATH is set to the combined values. When reading *java2cp1..8*, the search for names stops as soon as a logical name is undefined. In addition, if *java2cp1* is defined in the process logical names table, any *java2cp#* entries in the system logical names table, that are set via configuration, are ignored. After the environment files are processed, the logical names table is queried again. If the logical name *classpath* is defined, that value replaces any existing value of CLASSPATH in the C run-time environment. You can completely override any setting of CLASSPATH done in the environment files by defining *classpath* on the command line.

**PATH** The 4690 path is blank delimited. This is converted to a semi-colon delimited list because the VisualAge C runtimes expect it in this form.

**TMP** This is set to ADX\_UDT1: on the controller and R::ADX\_UDT1: on terminals. The value for TMP is set before the environment data files are processed, so it can be overridden, if needed.

## Environment File Processing

The file ADXENV.DAT, which contains global settings to be used for all applications, is read first. Then, controller or terminal specific settings are read from ADXENVC.DAT or ADXENV.T.DAT respectively. If an environment variable is set multiple times (either later in the same file or in the second of the two files that are read in), then the last setting is the one used.

Environment strings can be imbedded in the value part of the string by surrounding the name of an environment variable in the percent (%) sign. For example, if the environment file contains:

```
OS=c:\win
OS=%OS%\crash
```

The final value of OS will be "C:\WIN\CRASH".

An undefined environment name contained in percent (%) signs is replaced with an empty string. In order to imbed a percent (%) sign in an environment value, replace it with "%%". Unmatched percent (%) signs are ignored. Because the environment name can contain a percent (%) sign, it can lead to names that cannot be expanded. In the example:

```
E1=
E2=
E1%E2=5
TEST1=%E1%E2%
TEST2=%E1%%E2%
```

TEST1 will contain the value "E2%" and TEST2 will be blank (and therefore be undefined).

In addition to using the percent sign (%) to refer to other environment variables, you can also use it to refer to logical names. To do this, precede the logical name with an exclamation point (!) and surround the name in percent signs. For example:

```
MY_PROGRAM = %!adx_ipgm:%
MY_DATA    = %!adx_idt1:%
OS_VERSION = %!version%
```

The search order for the logical name is the process logical name table and then the system logical name table. If the name is present in both tables, the value in the process logical name table is used. If the logical name is not defined, it is replaced with an empty string. Wildcard characters are not allowed in the string.

When the logical name path is queried, it is converted from a blank delimited list of directory names to a semi-colon (;) list of directory names. C run-time functions assume that elements of a path are separated by semi-colons.

Logical names ending in colons (:) are automatically expanded by the operating system even when the files are opened using the C run-time functions.

Other rules involving environment variables on 4690 are:

- White space at the beginning and end of each line is ignored as well as white space to the left and right of the (first) equal (=) sign. That is, neither environment names nor values can contain leading or trailing spaces. Embedded white space is retained.
- If an environment variable is set to blank (that is, the line ends with the equals sign), the environment variable is removed from the environment. Querying its value using *getenv()* results in a NULL value being returned.
- The first carriage return or line feed encountered is considered the end of the line.

- Lines beginning with an equal (=) or semicolon(;) are considered comments and are ignored.
- Environment names can contain any ASCII character with a value greater than 0x20 (a space) except an equal sign.
- Environment values can contain any ASCII character with a value of 0x20 (a space) or higher.
- ASCII letters in environment names are converted to uppercase before being stored or used. The case of the values is not changed.
- Environment variables cannot start with a semicolon (;) left bracket ([), or an exclamation point (!).

## Overrides for Specific Machine IDs or Programs

The environment files are similar to Windows INI files in that they can be divided into named sections. A section is defined by putting the section name on a line by itself in square brackets ([ ]). Section names in the global file (ADXENV.DAT) are ignored. However, they can be used in the controller or terminal files to set up environment variables that are specific to a given controller or terminal. Sections can also be used to setup environment variables that are unique to a particular program.

Section names consist of a list of terminal or controller IDs that are optionally followed by a colon and a list of program names. The entries in each list are separated by either blanks or commas (,). An asterisk (\*) in a list matches any terminal or controller ID or process name.

When a particular program reads the environment file and a section name is found, any environment settings within that section are processed only if its controller or terminal ID is in the machine ID list and its program name is in the program name list. If the program name list is not present, the section is processed for any 32-bit VisualAge program running on the given controller or terminal. For example,

- The section names "[JB]" and "[JB:\*)" are processed for all programs running on the JB controller.
- The section named "[\*:hello,goodbye]" is processed for the programs "hello" and "goodbye" on any machine.
- The section "[007:BOND]" is only processed by the program "BOND" on terminal 007.

**Note:** Empty section names "[ ]" and section names without preceding text or following the colon "[:]", "[\*:]", "[\*:\*)" do not match anything and are never processed.

Keyword=value pairs that occur before the first section are considered to be in the default section and apply to all controllers and terminals. To resume adding entries common to all controllers or terminals, create a section named "[\*]", or "[\*:\*)".

Other rules used when processing sections are:

- A section ends at the beginning of the next section definition or at the end of the file.
- Case and leading or trailing spaces are not significant in the name of a section.
- Anything after the first right bracket (]) on a section heading is ignored.

## Special Commands

Environment files can contain special commands. Special commands are indicated by lines starting with an exclamation point and the command name. Commands are processed only if they are in the default section, or if the name of the section where they appear matches the current machine ID and program name. If an unrecognized command is encountered, it is ignored without reporting an error. Percent signs in the command parameters are treated as-is (they are not used to perform environment variable expansion as when setting environment variables). The following special commands are available:

**!include filename** — This command includes the contents of the given file into the environment. The file is processed as though its contents were inserted into the current environment file at the location of the include command. Section names are allowed in the included file. However, once the file is completely processed, the section name from the original file is restored. An environment file can include as many files as required. If the file name given to the **!include** command does not have a

fully-qualified path, the current directory is searched. An included environment file may include another environment file. However, this inclusion is limited to two levels deep and a total of three files open at once (the original file and two included files).

**!lookup** *logical\_name* ... — This command imports a number of logical names into the C run-time environment at one time. The **!lookup** command is followed by a list of logical names. Each name in the list should be separated by a blank or comma. The process logical name table and then the system logical name table is searched for each name in the list. If the logical name is defined, it is added to the C run-time environment (with the value in the process logical name table taking precedence). If the logical name is not defined, it is ignored (any existing environment variable of the same name is not removed). If the logical name contains wildcards, then all matching logical names are imported. Once a logical name is imported with this command, it is accessed just like any other environment variable.

**!lookup\_p** *logical\_name* ... — This command is identical to the **!lookup** command except that only the process logical name table is searched for the logical name.

**!lookup\_s** *logical\_name* ... — This command is identical to the **!lookup** command except that only the system logical name table is searched.

---

## Special Environment Variable Settings

### Disabling Buffering of Standard Streams

By default, the standard output streams, stdout and stderr, are buffered. Any output sent to them is not immediately written to the file. When output is being sent to a device, such as the console, the output is line buffered. In this mode, data is flushed, or written, to the file only when end-of-line characters are written. When the standard streams have been redirected to a file, the C run times detect this and do file buffering on the output. All output is stored in a buffer and written to the file only when the buffer is full. The default buffer size is 4 Kb. If your program traps, the process ends without doing extensive clean up and any data still in the buffer is lost.

Similar to other streams, the buffering strategy of the standard output streams can be changed by calling the *setvbuf()* function from within your program. However, if you are attempting to debug a program in an existing program, it might not be advisable or possible to change the code. For this reason, the C run times support an environment variable to control this. To use the environment variable, add the line `CRT_DISABLE_STD_BUFFER=on` to your environment settings. See “C Run-time Environment Variables on 4690” on page 668 for more information on environment settings. Enabling this setting is equivalent to calling *setvbuf()* with the `_IONBF` flag for both stdout and stderr. If the environment variable is set to a value other than *on*, the buffering behavior is not changed. Any additional changes to the standard output streams are not affected by this setting.

---

## Using Locales and iconv functions

The VisualAge compiler provides several functions for converting data that is encoded in one code set to another code set. In addition, the compiler allows the setting of run-time locale information. The VisualAge online help explains the use of these functions in more detail.

All locale information used by *setlocale* and the *iconv* functions is assumed to exist in the directory referred to by the *LOCPATH* environment variable. On the Windows NT operating system, this directory is set to `D:\IBM CXXW\LOCALE` (where D: is the drive on which the compiler is installed). On the 4690 operating system, the directory is user-defined. However, if terminals are accessing the directory using RIOAM, the name of the directory is limited to two characters (for example, `C:\LC`).

For controller applications, set the *LOCPATH* variable to the local directory name containing the locale information. In this example, you would have the following entry in the `ADXENVC.DAT` file.

```
LOCPATH=c:\lc
```

Similarly for terminals, set the variable to the proper remote directory on the controller containing the locale files. In the following example, you would have the following entry in the ADXENV.T.DAT file:

```
LOCPATH=r::lc
```

**Note:** In this case, the directory where the iconv and uconvtab directories are located can have no more than two characters in its name because of the path name limits of RIOAM. You also cannot have more than one directory between the root directory and the iconv and uconvtab subdirectories. These restrictions do not apply if you are using NFS to access the locale files on the controller.

## Using the iconv functions

To use the iconv character conversion functions on 4690, you must copy several files from your Windows NT installation of VisualAge C++ 3.6.5. If you installed VisualAge into directory D:\IBM CXXW, the conversion files are located in the directory D:\IBM CXXW\LOCALE. Within this directory are many subdirectories, including iconv and uconvtab. Create a directory on your 4690 system (for example, C:\LC) and copy the iconv and uconvtab subdirectories and their contents into this directory. The two .DLL files in the C:\xxxx\ICONV subdirectory on the 4690 system must be replaced with the copies of these files from the VisualAge development .ZIP file (VADEVENV.ZIP), which is located on the 4690 Optionals.

**Note:** You do not need to copy all of the files in the *uconvtab* directory. You can copy only the files needed for the specific code set conversions you want. Refer to the VisualAge documentation for more information on the character converters supplied by the compiler as well as information on creating new ones.

## setlocale and LCL files

By default, the C run-times support the "POSIX C" locale. This locale is standard for all ANSI-conforming and POSIX-conforming compilers. You can accept the default or you can use the *setlocale* function to set the locale to one of the locales supplied with the compiler. Locale information is stored in .LCL files, which are .DLL files saved with the .LCL extension. The VisualAge compiler contains many locale files located in the D:\IBM CXXW\LOCALE directory. The compiler documentation describes the naming conventions of these files. As with the iconv functions, you only need to copy the locale files you intend to use.

In order to use the locale files on a 4690 system, the files must be copied to your 4690 controller. If the locale name passed to the *setlocale()* function contains a path name, that path is used to locate the .LCL file. Otherwise, the compiler uses the value of the *LOCPATH* environment variable and the name of the locale to generate the directory path and file name of the locale file. If you use the same file and directory names when copying the contents of the compiler's locale directory to your 4690 system, you do not need to put the directory name in the locale name passed to *setlocale*.

---

## Running Your Application

To run your application, copy the application to your 4690 system and run it. If you use ftp to transfer the files, make sure you transfer the files in a binary format. When running applications from the command line, you do not have to type the .386 extension. However, if you try to run an application with a .286 extension and there is an application with the same name but with a .386 extension in the same directory, the command shell will run the .386 application unless you explicitly type in the extension.

---

## Run-time Exception Handling

When operating system exceptions are generated in 4690 OS V2 or higher, they are handled by 32-bit user code similar to the Structured Exception Handling (SEH) performed by Windows NT and OS/2 operating systems. These exception handlers are chained together using data on a thread's stack. Therefore, each function can have its own handler. The C run-time code registers a default exception handler to map some types of operating system exceptions into C signals (for example, SIGSEGV). The processing of C++ exceptions is also performed by the C run-time exception handler.



Normally, when the controller or terminal “application dump flag” is turned on, any system exception causes an operating system dump. However, in order for the C run-time library code to process system exceptions, it must request that the OS notify it whenever an exception occurs. Once a process has registered with the OS to receive exceptions, any exceptions generated for that process do not cause a dump. Instead the exception is sent to the process' exception handling routine.

**Note:** Dumps are only taken for “system” level exceptions such as page faults and divide by zero exceptions. Dumps are not taken for exceptions thrown by C++ code.

For example, if a program generates a page fault exception, and there is no SIGSEGV signal handler registered, then the C run-time prints a message indicating an exception occurred and ends the process. This behavior is acceptable in a development environment, but it is probably not correct in a production environment where the dump flag is turned on and is used for problem determination. Because of this, the C runtime only prints out a message when the dump flag is on. When the dump flag is off, the C runtime still attempts to handle the exception. If none of the registered exception handlers process the exception, the exception is sent back to the OS, which forces a dump.

In some cases, programs are written to expect and properly handle exceptions. Because the default behavior (based on the setting of the dump flag) is not acceptable in this case, there is a method to override this behavior. This is done by setting the C environment variable `CRT_EXCEPTION_TRAPPING` to one of the following values. If the environment variable is not set or is set to an unknown value, then the value “default” is used.

*on* — When the environment variable is set to this value, exceptions are always handled by the run times. Exceptions are passed to the structured exception handlers and C signal handlers as appropriate. If the exception is not handled, then the process ends.

*off* — When the environment variable is set to this value, exceptions are never handled by the run times. Structured exception and C signal handlers are not called. When an exception occurs, the OS automatically ends the process if the dump flag is off, or dumps if the dump flag is on.

**Note:** If `CRT_EXCEPTION_TRAPPING` is set to off, the Java 2 JVM does not work properly. The JIT depends on having exceptions enabled. Do not use this setting with Java 2.

*try* — When the environment variable is set to this value, the C runtime attempts to handle the exception by calling all of the structured exception handlers on the thread's stack. If none of the exception handlers process the exception, the exception information is passed back to the OS. If the dump flag is off, the OS ends the process. If the dump flag is on, the OS forces a dump.

*default* — When the environment variable is set to this value, or is not set at all, the default behavior described above occurs. That is, when the dump flag is on, the behavior is equivalent to the *on* setting. When the dump flag is off, the behavior is equivalent to the *try* setting.

**Note:** When the C run time first initializes, the “default” behavior is used. The environment variable setting only has an effect after the environment is successfully read in during initialization. Therefore, if the dump flag is on and the C run time encounters an exception before the environment is processed, a dump occurs. Any global C++ objects are constructed after the environment is read, so any exceptions generated during their construction is affected by the setting of the environment variable.

## JNI-Specific Information

The C run time exception handler is only registered for threads initialized by the current C run time. JNI code placed in .DLLs does not share the same C run time as the Java Virtual Machine (JVM). Therefore, you must indicate to the system that it must register an exception handler for each JNI function in your .DLL. To do this, use the C handler pragma as described in the VisualAge online documentation. When using this function, you only need to include the first parameter for the pragma (the function name). The second parameter, the exception handler, should be left blank to ensure that the default C run time exception handler is registered.

---

## Restrictions, Limitations, and Differences

This section contains the restrictions, limitations, and differences of creating and using 32-bit applications using VisualAge C/C++ within the 4690 OS V2 or higher environment.

**Note:** The model 2 printer driver does not support 32-bit applications.

### Platform Restrictions

Only 386 and above processors are supported. If the system on which you are running the code does not have a floating point processor (FPU), an exception is generated if you attempt to use floating point numbers or call run-time functions that use floating point numbers. Any run-time routine that accepts or returns a float, double, or long double value will typically use FPU instructions. In addition, other functions that use FPU instructions include:

- `fpreset()`
- `fpuinit()`
- `control87()`
- `clear87()`
- The `printf()` and `scanf()` family of functions (when floating point type characters are used in the formatting string).
- Dividing two "signed long long" values (`_divi64`)
- Any functions dealing with `time_t`, which is defined as a double. These also include functions such as `utime()` and `stat()` that work with structures containing elements of type `time_t`. See "time\_t Changes" on page 674 for more information on using `time_t` on systems without an FPU.

### time\_t Changes

4690 OS V2 or higher allows the use of `time_t` on a non-FPU system. The `INCLUDE\NONFPU` directory of the development environment contains a set of replacements for the VisualAge C run-time header files working with `time_t`. These files define `time_t` as a "long long" and provide a set of functions for working with the different type. The files use compiler pragmas to allow the differences to be transparent to the application. The main difference is the type of `time_t` variable. Applications compiled with the updated headers can also be used on systems that have an FPU.

In VisualAge C/C++ version 3.5, `time_t` was defined as a "long". In VisualAge C/C++ version 3.6, `time_t` has been defined as a "double". The change in run times was made to ensure the compiler was compatible with ANSI standards and to avoid future `time_t` value problems. Many existing applications have made assumptions regarding the type or size of `time_t`. Because of this, VisualAge version 3.5 headers have been provided in the `INCLUDE\OLDTIME` directory of the development environment. Using the old headers may cause additional overhead when using time related functions, so it is recommended that you convert your code to use the "double" form of `time_t` if possible.

To use either of these sets of headers, compile each of your source files so that the directory name (`INCLUDE\NONFPU` or `INCLUDE\OLDTIME`) is listed before the VisualAge C include directory. To do this, place the directory name first in the `INCLUDE` environment variable or by adding it as the first directory in the `/I` compiler option. All files in your application must be compiled with the same set of headers to ensure you get consistent and predictable behavior.

### Differences Between Windows NT and 4690 OS

File names on Windows NT systems are treated differently from file names on the 4690 OS V2 or higher system. These differences include:

- 4690 OS V2 or higher performs logical name expansion when file names contain logical names that end in a colon (:).
- Standard FAT formatted disks use the 8.3 naming convention, and have a total of 127 characters in the path.
- Files accessed through RIOAM have a 26-character limitation.



- A double slash or backslash in a file name is ignored by Windows NT, but within 4690 causes the operating system to restart the path at the root (that is, it causes everything between the double slash and the drive or node designator to be ignored).

File naming conventions within the 4690 operating system are different depending on whether you have an FAT or VFS system. See Chapter 2, “Managing files,” on page 11 for rules and restrictions on naming files on both FAT and VFS systems.

In addition, several functions behave differently on 4690 than they do on NT. The following tables lists these differences.

Function	Difference
<code>exec()</code>	On Windows NT files that are open when an <code>exec</code> call occurs remain open in the new process. This is not true for 4690 OS. A C program cannot determine the file handles used by the operating system.
<code>rmdir</code>	The current directory of a program can be removed if the directory is empty. 4690 OS V2 or higher uses logical names to track the current directory of a program and does not keep the directory "open" as in Windows NT.
<code>remove</code>	If only one process has a file open, that process can remove the file while the file is still open. The file is automatically deleted when your process ends. This is not allowed on Windows NT systems.
<code>spawn/exec</code>	If the path name given does not contain an extension, the extension <code>.386</code> is appended instead of a <code>.EXE</code> extension.
<code>stat/fstat</code>	When calling these functions on a file, the <code>st_mtime</code> , <code>st_atime</code> , and <code>st_ctime</code> are all set to the same value. When calling <code>stat</code> on the root directory of a drive (for example, <code>C:\</code> ), the time values will be set to the beginning of the C run time 'time epoch' (midnight on January 1, 1970).
<code>_beginthread</code>	Stack size for <code>_beginthread</code> is limited to 508Kb.
<code>main/exit</code>	Returning negative values from <code>main()</code> or from <code>exit()</code> in the 4690 operating system causes the command shell to print a message corresponding to that value. Returning negative values from terminal applications may cause the terminal loader to report an error and reload the default application. Typically, these errors will be cryptic and not related to the error. This is also the case if you declare <code>main</code> as <code>void</code> . The code that calls <code>main</code> in the run time expects a return value. If <code>main</code> is declared as returning <code>void</code> , the value returned to the run times will be undefined. It is recommended that you do not return negative return codes and that you do not declare <code>main()</code> as returning <code>void</code> .
<code>system()</code>	Applications run using the <code>system()</code> function do not inherit the environment of the calling program. Only <code>.386</code> applications written using the VisualAge run times and called using the appropriate <code>exec/spawn</code> function will inherit the environment. The <code>system()</code> function is documented as returning the return value from the command processor if the command processor is successfully called. When used to run an executable program, the 4690 command processor returns the lower 16 bits of the exit code of that program (the upper 16 bits are not returned). In addition, if the command processor is given a non-valid command line or if the command processor cannot find the given program, the processor returns 0 with no indication of an error other than a message printed to the standard output file or console.
<code>argv[0]</code>	<code>.286</code> applications and <code>.386</code> applications written without using the VisualAge run times will ignore the value for <code>argv[0]</code> passed by <code>spawn/exec</code> . These programs typically get their value for <code>argv[0]</code> from the operating system.
<code>chdir()</code>	The documentation for <code>chdir()</code> indicates that it will not change the current drive (if the directory name passed in contains a drive letter). However, under NT, it does change the current drive. This “feature” has been ported over to 4690 as well.

Function	Difference
SIGINT/SIGBREAK	<p>No distinction is made between the Ctrl+C or Ctrl+Break keys in 4690. If one of these key combinations is pressed, the SIGINT signal is raised. If the signal is not handled, the SIGBREAK signal is then raised.</p> <p><b>Note:</b> The signal might be raised on any thread. Typically, it will be the last thread to do any console I/O.</p>
cwait	<p>This function works only with child processes created from the spawn family of functions using the P_NOWAIT flag. In addition, a thread can only use cwait to wait on child processes that it created itself; threads cannot wait on child processes created by another thread. The return status for cwait (which is placed in the location pointed to by the first parameter) is set differently under 4690. In 4690, it is not possible to tell whether a process ended abnormally. However, certain negative values of the exit code can indicate an abnormal termination. Therefore, when the exit code of a child process indicates it did not end abnormally, the lowest-order byte of the variable pointed to by stat_loc is set to 0 and the next highest-order byte is set to the lowest-order byte of the exit code of the process. If the exit code of a child process indicates it may have ended abnormally, the lowest-order byte of the variable pointed to by stat_loc is set to the low byte of the exit code and the next higher-order byte is set to 0.</p>
spawn	<p>When called with the P_NOWAIT flag, this function will cause an event bit to be used. This event bit is later used to wait for the process to end through the cwait() function. Because there are only 31 event bits per thread, a maximum of 31 outstanding asynchronous calls may be outstanding at a time, including asynchronous reads and writes as well as child processes spawned with the P_NOWAIT flag. The event bit used by spawn is released automatically if the parent process is waiting on the child (through cwait()) when the child process ends. If the child process ends before the cwait() function is used to wait on it, the event bit is also released when cwait() is called and before any other asynchronous child processes are created.</p>

## 4690–Specific Behavior

When a .DLL contains global C++ objects, the constructors and destructors for these objects are run by the C run-time initialization and termination routines of that DLL. This also applies when you write your own DLL initialization or termination routines. The following functions cannot be performed during DLL initialization or termination and will fail with an error code if attempted:

- Starting a process
- Loading DLLs and unloading DLLs

**Note:** Although the current process will receive an error code when performing these functions, other processes will be affected as well. If any other process attempts to start a program or load or unload a .DLL while the dynamic DLL initialization or any DLL termination routines are running, that process will be blocked until those routines end. Therefore, do not perform excessive or time consuming amounts of work during DLL initialization or termination routines.

## Terminal Differences

4690 OS terminals have no support for consoles. Therefore, using the standard handles (*stdin*, *stdout*, *stderr*) will not work. The run-times map the handles internally to invalid file handles so operations that use them will always fail. Functions that attempt to access standard handles will return error codes. These functions include, but are not limited to:

- getch
- getche
- cputs
- cgets
- cprintf
- printf
- getc
- gets

- `scanf`
- `kbhit`
- `perror`
- `putch`

To access files on the controller from a terminal, you must either mount a controller drive using Network File System (NFS) and use those drive letters, or you must prefix the file name with `R::` to use RIOAM. However, using RIOAM has restrictions, which include the subdirectory depth and path length. If a terminal has a local hard disk, it will be assigned the `C:` drive letter and files can be accessed on it.

**Note:** There is no access to diskette drives `A:` or `B:` on a terminal.

Functions that create new processes (`exec/spawn/system`) are not intended to be used on terminals; use `ADX_TCHAIN` instead. (See Chapter 19, “Designing Terminal Applications With C Language,” on page 397 for more information on the `ADX_TCHAIN` function). In particular, when these functions are used, the terminal control process will not know about the newly created process, and you will not be able to stop the process from the terminal functions panel on the controller.

Creating directories through RIOAM using the `mkdir()` function is not supported. If you try to create a directory through RIOAM using the `mkdir` function, a file with the same name will be created instead.

Calling `stat()` on directories accessed using RIOAM returns results that are not valid; all fields except the directory attribute bit are undefined. The results from `stat()` indicate only that it is a directory.

When an application starts, the current directory will be set to the root directory of the first valid drive letter. If there are no local, NFS, or RAM disks available, the current directory will be set to an empty string.

## Using JNI in 4690

Normally, when a Java program is run on a terminal, the program can call methods such as `System.out.println()` to write output to the terminal’s video display. Any output using these methods goes to the Java console (which is a different logical console than the one written to by non-Java programs). By using the invocation API, a C/C++ program can create a Java VM and run Java code.

However, the console assigned to the program by the operating system is the non-Java console. Therefore, any output created using the `System.out` and `System.err` Java objects does not appear on the panel as expected. In addition, any informational messages from the Java VM (such as configuration or classpath problems) are not displayed.

## Unsupported Features

4690 OS V2 or higher supports only the multi-threaded C run-time libraries. 4690 OS V2 or higher does not provide or support:

- Single-threaded and subsystem libraries
- System Object Model (SOM) Code
- Use of `C __thread` keyword

---

## Problem resolution

A program might not work properly or might cause other problems. Before contacting your support representative, refer to Table 62 for common problems and how to resolve them.

Table 62. Problem resolution

Symptom	Solution
I am using C++ streams for program output. Output data is not being flushed as expected even though I have placed the newline character ("\\n") into the output stream. Previously, when using C output functions, such as printf, this worked fine.	<p>You must use output an endl or explicitly flush the stream to have buffered output displayed. Unlike C output streams, C++ streams are not buffered using simple line buffering. As referenced in the compiler programming guide, a stream is flushed implicitly in the following situations:</p> <ul style="list-style-type: none"><li>• The predefined streams cout and clog are flushed when input is requested from the predefined input stream (cin).</li><li>• The predefined stream cerr is flushed after each output operation.</li><li>• An output stream that is unit buffered is flushed after each output operation. A unit-buffered output stream is a stream that has ios::unitbuf set.</li><li>• An output stream is flushed whenever the flush() member function is applied to it. This includes cases where the flush or endl manipulators are written to the output stream.</li><li>• The program terminates.</li></ul>

---

## Chapter 23. Using the Enhanced Security Passwords API

---

### Introduction to enhanced passwords

4690 OS Version 5 added application programming interfaces that allow more direct control over user authentication and security management. This chapter describes this function and how to use it.

#### Notes:

1. The options included in this product can help your company address the PCI DSS requirements.
2. The customer is responsible for evaluation, selection, and implementation of security features, administrative procedures and appropriate controls.
3. PCI DSS is Payment Card Industry Data Security Standards.

---

### User ID rules

The user ID identifies an individual user. Rules for the user ID are:

- User IDs must consist of from 1 to 9 characters.
- Valid characters include uppercase English letters A–Z, lowercase English letters a–z, numbers 0-9, and special characters #, {, }, (, and ).
- User IDs are not case sensitive. All letters in an ID are converted to uppercase before being used by the system.
- User IDs must be unique.

A user ID value is also used to give names to *model records*. Model records are used as templates to create new user records. There is no password associated with a model record and a model record cannot be used to log in or authenticate to the system.

---

### Password rules

The password authenticates the user to the system. Users should not share or divulge their passwords.

Rules for passwords are:

- Passwords must consist of from 2 to 8 characters.
- Valid characters include uppercase English letters A-Z, lowercase English letters a-z, numbers 0-9, and special characters #, {, }, (, and ).
- Passwords are case-sensitive.
- The user ID cannot be included in the password.
- A minimum password length can be imposed through configuration. See “GetMinPasswordLength” on page 707 and “SetMinPasswordLength” on page 707.
- A password expiration duration (in days) can be imposed through configuration. See “SetExpires” on page 706 and the “GetExpires” on page 706.
- The number of days prior to the expiration date to present a prompt can also be configured. See “GetExpireWarning” on page 708 and the “SetExpireWarning” on page 708.
- The password can not be reused within four cycles.
- The password can not include a run of more than 2 repeated characters (for example, “AAA”).
- The password cannot include sequences of more than 2 sequential characters (for example, “456” or “cde”).
- The password must contain both a letter (either upper or lower case) and a number.
- (Optional) When you change a password, at least 4 characters must change. See “GetMinChange” on page 709 and “SetMinChange” on page 709.

- (Optional) The password must contain one each of the following characters (see “GetMultipleCharacter” on page 710 and “SetMultipleCharacter” on page 711):
  - A lowercase letter
  - An uppercase letter
  - A number
  - A special character

## Authorization attributes

The entire set of authorization attributes is represented as a string of 110 characters:

- The first 104 bytes of the string contain various permission fields. Each of the first 104 bytes will be set to **Y** (yes), if the corresponding permission is enabled or the action is allowed and to **N** (no), if the permission is not enabled or the action is not allowed. For example, if the Privileged Window Control field in the string is set to **Y**, the user can use Privileged Window Control. If it is set to **N**, the user cannot use Privileged Window Control. Refer to Table 63.
- Characters 105-107 are the three digit group number. The group number ranges from 002 to 254.
- Characters 108-110 are the three digit user number. The user number ranges from 001 to 254.

Compiler constants are provided to allow access to the various attribute values. In C and Java the permissions have the names of the format `ADX_CPW_ATTR_XXX`, where `XXX` is the type of permission being granted. If `XXX` starts with `RESERVED`, then this permission attribute is reserved and unused. The corresponding attribute character in the string should be set to **N** in this case.

The following is the complete attribute string for the master authorization record, which allows all valid permissions, broken into two 55 character strings:

```
YYYYYYNNYYYYYYNNYYYYYYNNNNNNNNYYYYYYYYYYYYYYNNYYYYYY
YYYYYYNNNNNNYYYYYYNNYYYYYYNNYYYYYYNNNNNN002001
```

Table 63. Authorization attributes

Character order in string	Permission attribute	Character order in string	Permission attribute
1	MULTIPLE_APPLICATION_MODE	53	RESUME_STORE_TCC_CONTROL
2	PRIVILEGED_WINDOW_CONTROL	54	COMMUNICATION_FUNCTIONS
3	BACKGROUND_APPLICATION_CONTROL	55	ACTIVATE_MASTER_CONTROLLER
4	SYSTEM_CONTROL_FUNCTIONS	56	ACTIVATE_FILE_SERVER_CONTROLLER
5	AUXILIARY_CONSOLE_CONTROL	57	DUMP_TERMINAL_STORAGE
6	MULTIPLE_SIGNON_MODE	58	DUMP_CONTROLLER_STORAGE
7	RESERVED	59	SET_SYSTEM_MESSAGE_LEVEL
8	RESERVED	60	LOAD_TERMINAL_CONFIGURATION_DATA
9	ENABLE_LINK	61	RESERVED
10	DISABLE_LINK_NO_FORCE	62	RESERVED
11	DISABLE_LINK_FORCE	63	RESERVED
12	DISPLAY_LINK	64	RESERVED
13	ENABLE_LAN_FOR_LU_6_2_COMMUNICATIONS	65	KEYED_FILE_UTILITIES
14	DISABLE_LAN_FOR_LU_6_2_COMMUNICATIONS	66	DISPLAY_ALTER_FILE_DATA
15	DISPLAY_LAN_STATUS_FOR_LU_6_2_COMMUNICATIONS	67	RESERVED
16	RESERVED	68	RESERVED
17	RESERVED	69	DISTRIBUTED_FILE_UTILITIES
18	FILE_UTILITIES	70	FILE_COMPRESS_DECOMPRESS
19	INSTALLATION_AND_UPDATE_AIDS	71	STREAMING_TAPE_DRIVE_UTILITIES
20	PROBLEM_ANALYSIS_DATA_COLLECTION	72	OPTICAL_DRIVE_UTILITY
21	PROBLEM_ANALYSIS_REPORTS	73	SYSTEM_CONFIGURATION_SYSTEM_SETTINGS

Table 63. Authorization attributes (continued)

Character order in string	Permission attribute	Character order in string	Permission attribute
22	COMMAND_MODE	74	SYSTEM_CONFIGURATION_SECURITY_SETTINGS
23	RESERVED	75	SYSTEM_CONFIGURATION_JAVA_SETTINGS
24	RESERVED	76	FTP_USER_DEFINITIONS
25	ENABLE_CONTROLLER_RAM_DISK	77	ENHANCED_SECURITY
26	DISABLE_CONTROLLER_RAM_DISK	78	SSH_SECURE_REMOTE_LOGON
27	RESERVED	79	SSH_SECURE_FTP_LOGON
28	RESERVED	80	PASSWORD_SETTINGS
29	RESERVED	81	CHANGE_CONFIGURATION_DATA
30	RESERVED	82	REPORT_LEGACY_CONFIGURATION_DATA
31	RESERVED	83	CHANGE_INPUT_SEQUENCE_TABLE_DATA
32	RESERVED	84	REPORT_MODULE_LEVEL
33	DEACTIVATE_MASTER_CONTROLLER	85	APPLY_SOFTWARE_MAINTENANCE
34	DEACTIVATE_FILE_SERVER_CONTROLLER	86	BUILD_SOFTWARE_MAINTENANCE_CONTROL_FILE
35	LOAD_CONTROLLER_STORAGE	87	SYSTEM_MESSAGE_AUDIBLE_ALARM_FUNCTIONS
36	TERMINAL_FUNCTIONS	88	RESERVED
37	CONTROLLER_FUNCTIONS	89	SCAN_SYSTEM_LOG_DATA
38	SYSTEM_FUNCTIONS	90	FORMAT_SYSTEM_TRACE_DATA
39	TCC_FUNCTIONS	91	FORMAT_PERFORMANCE_DATA
40	MULTIPLE_CONTROLLER_FUNCTIONS	92	FORMAT_DUMP_DATA
41	ENABLE_TERMINAL_STORAGE_RETENTION	93	CREATE_PROBLEM_ANALYSIS_DISKETTE
42	DISABLE_TERMINAL_STORAGE_RETENTION	94	RESERVED
43	SET_SYSTEM_DATE_AND_TIME	95	RESERVED
44	DISPLAY_TERMINAL_STATUS	96	RESERVED
45	START_TERMINAL_APPLICATION	97	START_TRACE_DATA_COLLECTION
45	STOP_TERMINAL_APPLICATION	98	START_PERFORMANCE_DATA_COLLECTION
47	JAVA_APPLICATION_FUNCTIONS	99	STOP_TRACE_DATA_COLLECTION
48	LOAD_TERMINAL_STORAGE	100	STOP_PERFORMANCE_DATA_COLLECTION
49	RESERVED	101	RESERVED
50	DISPLAY_CONTROLLER_STATUS	102	RESERVED
51	ALLOW_STORE_CONTROLLER_BACKUP	103	RESERVED
52	PREVENT_STORE_CONTROLLER_BACKUP	104	RESERVED

## User-defined attributes

In addition to the attributes described in “Authorization attributes” on page 680, the user is allowed to define 8 additional attributes, each of which can be set to either **Y** or **N**. The use of these attributes must be coordinated by the user with all other applications running on the machine.

## Error codes

The possible return codes are:

**ADX\_CPW\_ERR\_ID\_NOT\_FOUND** — 0x80E70001 — The given ID was not found.

**ADX\_CPW\_ERR\_ID\_NOT\_AUTHORIZED** — 0x80E70002 — The ID and password are valid, but are not authorized to make changes to security settings.

**ADX\_CPW\_ERR\_PASSWORD\_DOES\_NOT\_MATCH** — 0x80E70003 — The ID and/or password are not valid (that is, authentication failed for this ID and password pair).

**ADX\_CPW\_ERR\_PASSWORD\_EXPIRED** — 0x80E70004 — The user’s password has expired. The user cannot log in or perform other actions until the password is changed.



**ADX\_CPW\_ERR\_ID\_ALREADY\_EXISTS** — 0x80E70006 — The given ID already exists and must be deleted before a record with this ID can be created or copied. Note that lowercase letters in the ID are converted to uppercase when ID records are created.

**ADX\_CPW\_ERR\_PASSWORD\_SAME\_AS\_PREVIOUS** — 0x80E70007 — The requested password is the same as a previous password. A different password must be selected.

**ADX\_CPW\_ERR\_PASSWORD\_CONTAINS\_ID** — 0x80E70008 — The password contains the user's ID. A different password must be selected.

**ADX\_CPW\_ERR\_PASSWORD\_CONTAINS\_SEQUENCE** — 0x80E70009 — The password contains a sequence of more than 2 sequential characters (for example, "DEF" or "567"). A different password must be selected.

**ADX\_CPW\_ERR\_PASSWORD\_CONTAINS\_REPETITION** — 0x80E7000A — The password contains a string of more than 2 repeated characters. A different password must be selected.

**ADX\_CPW\_ERR\_TOO\_FEW\_CHARACTERS\_CHANGED** — 0x80E7000B — Too few characters in the new password have been changed. A different password must be selected.

**ADX\_CPW\_ERR\_PASSWORD\_MUST\_CONTAIN\_CHAR\_OF\_EACH\_TYPE** — 0x80E7000C — The password does not contain at least one uppercase letter, one lowercase letter, one numeric character, and one special character. A different password must be selected. This error is returned when the "multiple character" option is enabled and the password does not meet the requirement of containing a character of each type.

**ADX\_CPW\_ERR\_RECORD\_LOCKED\_BY\_ANOTHER\_SESSION** — 0x80E7000D — The user record is currently locked and cannot be accessed. This error is also returned when the current session has the record locked and a request requiring the record to be unlocked is made. This includes cases where a program attempts to lock a record twice.

**ADX\_CPW\_ERR\_SESSION\_TIME\_OUT** — 0x80E7000E — The user's session has been active for too long and has timed out. All API calls made with the same session ID will continue to return this error until the Cancel() API is called. The application must create a new session and redo any changes before time expires.

**ADX\_CPW\_ERR\_INVALID\_BUFFER** — 0x80E7000F — This error is returned when there is a buffer error. Buffer errors includes cases where a string, buffer, or other pointer passed to the API is not valid or the pointer is not valid for the given for the specified length. It is also returned in cases where the provided buffer is not the correct size to return the correct data, as with the GetIDs() and GetModelIDs() APIs.

**ADX\_CPW\_ERR\_SESSION\_LIMIT\_EXCEEDED** — 0x80E70010 — Only a limited number of sessions can be created at the same time. This error code indicates that there are too many active sessions already in existence.

**ADX\_CPW\_ERR\_INVALID\_SESSION\_ID** — 0x80E70011 — The session ID is not valid. This error is also returned on any API call that requires a session number, if the call is made when the enhanced security service is unavailable.

**ADX\_CPW\_ERR\_INVALID\_DATA** — 0x80E70012 — The data provided by the API is not valid. This error is returned when the provided data values or buffer contents are not valid or are out of range for the given API.

**ADX\_CPW\_ERR\_RECORD\_NOT\_LOCKED\_BY\_THIS\_SESSION** — 0x80E70013 — The record cannot be modified because it is not locked. Lock the record and retry.

**ADX\_CPW\_ERR\_INVALID\_TARGET\_ID** — 0x80E70014 — The requested target ID on a create or copy call is not valid.

**ADX\_CPW\_ERR\_INVALID\_PASSWORD** — 0x80E70015 — The new password contains characters that are not valid.

**ADX\_CPW\_ERR\_PASSWORD\_TOO\_SHORT** — 0x80E70016 — The new password is too short.

**ADX\_CPW\_ERR\_SERVICE\_NOT\_AVAILABLE** — 0x80E70017 — Enhanced Security is not configured and available.

**ADX\_CPW\_ERR\_NOT\_ACTING\_MASTER** — 0x80E70018 — This controller is not the acting master or cannot contact the acting master. Sessions can only be created and used on the acting master or from a node



that can contact the acting master. This error is also returned when the ChangePassword API is called and the acting master controller cannot be contacted. Passwords can only be changed when the acting master is available.

**ADX\_CPW\_ERR\_PASSWORD\_FILE\_FULL** — 0x80E70019 — The password file is full, it contains the maximum 2048 entries. No more operator IDs or models can be added.

**ADX\_CPW\_ERR\_UNEXPECTED\_IO\_ERROR** — 0x80E7001A — An unexpected file I/O error has occurred while processing this request.

**ADX\_CPW\_ERR\_UNSUPPORTED\_FOR\_MODELS** — 0x80E7001B — The ID is a model ID. The passwords for model records cannot be changed and model IDs cannot be used to validate a password.

**ADX\_CPW\_ERR\_MIN\_PW\_LENGTH\_MISMATCH** — 0x80E7001C — This error is returned when the setting change being made would result in a setting that is not valid for minimum password length. Normally, the valid range for the minimum password length setting is from 2 to 8. However, two optional requirements can affect the minimum password length:

- If the option to require a password to contain an upper case character, lower case character, numeric character, and special character is selected (the multiple character option), the minimum password length is four.
- If the option to require a four character change in the password is selected, the minimum password length is also four.

**ADX\_CPW\_ERR\_ATTRIBUTE\_MISMATCH** — 0x80E7001D — This error is returned when an attribute change being made would result in a set of attributes that is not valid (for example, if a reserved attribute is being enabled). Also, some attributes are prerequisites of others. For example, the attribute allowing configuration to be run (ADX\_CPW\_ATTR\_SYSTEM\_CONFIGURATION\_SYSTEM\_SETTINGS) cannot be enabled unless the attribute allowing access to configuration and update aids (ADX\_CPW\_ATTR\_INSTALLATION\_AND\_UPDATE\_AIDS) is also enabled.

**ADX\_CPW\_ERR\_PASSWORD\_MUST\_CONTAIN\_ALPHA\_AND\_NUMERIC** — 0x80E7001E — The password does not contain both an alphabetic character and a numeric character. A different password must be selected.

**ADX\_CPW\_ERR\_UNSUPPORTED\_FOR\_MASTER\_RECORD** — 0x80E7001F — The operation is not supported for the master record of the file. Currently, the master record cannot be deleted.

**ADX\_CPW\_ERR\_UNAUTHORIZED\_ATTRIBUTE\_SET** — 0x80E70020 — This error is returned if a user tries to enable an attribute that is disabled in its own record. This error is also returned if a user tries to set the Group Number or User Number to a value less than in its own.

**ADX\_CPW\_ERR\_AUTHORIZATION\_LEVEL\_ERROR** — 0x80E70021 — This error is returned when a user attempts to lock a record that has an authorization level greater than its own. This error is also returned when a user tries to copy/create a record with an authorization level greater than in its own. It is also returned when a user tries to set a record's authorization level higher than its own.

**ADX\_CPW\_ERR\_PASSWORD\_SETTINGS\_AUTHORIZATION\_ERROR** — 0x80E70022 — This error is returned if the user tries to set global attributes and does not have *Enhanced Security Password Settings* enabled in the system attributes.

**ADX\_CPW\_ERR\_TIMEOUT** — 0x80E70023 — This error is returned when an API request times out before completion.

---

## Sessions

Most of the Enhanced Password functions require a session to accomplish. Only users that are authorized to make changes to security settings are allowed to create a session. The functions that do not require a session include functions to validate an ID or password, change a password, and retrieve attributes.

Functions that use a session number cannot update a record until the record is locked, but unlocked records can still be read. However, when a session owner locks a record, all other attempts to read that record (even from another session) fail until the session ends. After all changes within a session are complete, the changes can either be committed to the database or canceled.

Currently only one session can be active at a time.

---

## Application Programming Interfaces

This section describes the APIs for C, CBASIC, and Java. The Java programming interfaces are described more fully in javadoc html files. These files can be found in the file `enhpwdoc.jar` which is located in the 4690OPT directory of the installation CD.

**Note:** For V6R3, several Enhanced Security API calls have been updated to support LDAP directory IDs. For information about these updates, see *Updates to the Enhanced Security API4690 OS User's Guide*.

Any function, except `IsAvailable`, returns the error `ADX_CPW_ERR_SERVICE_NOT_AVAILABLE` if Enhanced Security is not currently enabled and operational.

### **For C (both 16-bit and 32-bit interfaces):**

- Any pointers declared with the “const” modifier can not be modified by the API. All other data referenced with pointers can be modified.
- A header file is provided that contains function prototypes and constant definitions for these routines.
- The 16-bit link library (`adxcpwcl.l86`) and header file (`adxcpwcc.h`) are located in the 4690OPT directory of the installation CD. The l86 file is designed for large model controller applications.
- The 32-bit link library (`adxcpwcl.lib`) and header file (`adxcpwcc.h`) are located in `vadevenv.zip` along the other 32-bit C development files. The ZIP file is located in the 4690OPT directory of the installation CD.
- Incoming pointer parameters that do not otherwise have a length, like *id* and *password*, must be null terminated strings.
- Attribute buffers (which do have a buffer size, typically called *buffsize*) are NOT null terminated.

### **For CBASIC:**

- The header file `adxcpwcc.j86` contains variable and subroutine declarations.
- The names for the API calls, error codes, and attribute field indices in CBASIC have underscores replaced with periods.
- Variables are implemented as functions that return the corresponding error code, attributes index, or other value. Because of linker limitations, the maximum symbol name is limited to 40 characters. Some of the longer error and variable functions are truncated to 40 characters (or 39 if the 40th character is a period).
- The first parameter of each subroutine is the return code where the result of the call is returned.
- The link library `adxcpwcl.l86` contains code required to link an application using these APIs. (This is a large model library for controller applications.) These files are located in the 4690OPT directory of the installation CD.

### **For Java:**

- The Enhanced Password classes are in the `com.ibm.OS4690.security` package. When method names are listed in the following sections, the class name containing the method is listed in parenthesis.
- All methods described in following sections throw a `FlexosException` if an error is detected; the `getReturnCode()` method of this class can be used to retrieve the error code. This class is in the `com.ibm.OS4690` package.
- The Java classes for this API (and the `FlexosException` class) are provided in `javabin:os4690.zip`. You must add this file to your classpath to make use of these functions.
- The error code and attribute field index constants are contained in the class `com.ibm.OS4690.security.EnhancedSecurityConstants`.

## IsAvailable

IsAvailable checks to see if Enhanced Security is active and available for use.

### C interface:

```
int ADX_CPW_IsAvailable();
```

### CBASIC interface:

```
SUB Adx.Cpw.IsAvailable(rc) EXTERNAL
    INTEGER*2 rc
END SUB
```

### Java interface:

```
(com.ibm.OS4690.security.EnhancedSecurity)
boolean isAvailable();
```

### Return codes:

- 0 — Enhanced security is not available.
- >0 — Enhanced security is available. Currently this is always 1. Bit 0 of the return value will always be set. Other bits are reserved.

## GetCurrentUserID

GetCurrentUserID returns the ID of the current user in the buffer provided. If the user ID has been deleted since the user logged in or the user record is locked, this call fails with an appropriate error.

The user ID for a process that was not started by a specific user's login session (or a background process) is considered to be "BACKGRND". If this API is called from a background process and the ID "BACKGRND" does not exist, the error code ADX\_CPW\_ERR\_ID\_NOT\_FOUND is returned. Otherwise, this API functions as it would for a logged in user ID.

In C, the buffer must be large enough to hold the returned user ID plus a terminating null character; to allow for future growth; allocate 17 bytes total. The returned ID is not padded with blank characters.

### C interface:

```
long ADX_CPW_GetCurrentUserID(char *buffer, long buffsize);
```

### CBASIC interface:

```
SUB Adx.Cpw.GetCurrentUserID(rc, buffer) EXTERNAL
    INTEGER*4 rc
    STRING buffer
END SUB
```

### Java interface:

```
(com.ibm.OS4690.security.EnhancedSecurity)
String getCurrentUserID();
```

### Return codes:

- 0 — Success
- ADX\_CPW\_ERR\_SERVICE\_NOT\_AVAILABLE
- ADX\_CPW\_ERR\_ID\_NOT\_FOUND
- ADX\_CPW\_ERR\_RECORD\_LOCKED\_BY\_ANOTHER\_SESSION
- ADX\_CPW\_ERR\_INVALID\_BUFFER

## Validate

Validate validates an operator ID (*id*) and the password for the operator (*pw*). It does not require a session ID. When the number of days left until the password expires falls within the configured value of the “expire warning” setting (1-15), the number of days remaining is returned.

### C interface:

```
long ADX_CPW_Validate(const char *id, const char *pw);
```

### CBASIC interface:

```
SUB Adx.Cpw.Validate(rc, id, pw) EXTERNAL
    INTEGER*4 rc
    STRING id
    STRING pw
END SUB
```

### Java interface:

```
(com.ibm.OS4690.security.EnhancedSecurity)
int authenticateUser(String id, String password);
```

### Return codes:

- 0 — User ID and password are valid
- 1-15 — The user ID and password are valid, but will expire in the given number of days.
- ADX\_CPW\_ERR\_SERVICE\_NOT\_AVAILABLE
- ADX\_CPW\_ERR\_ID\_NOT\_FOUND
- ADX\_CPW\_ERR\_PASSWORD\_DOES\_NOT\_MATCH
- ADX\_CPW\_ERR\_PASSWORD\_EXPIRED
- ADX\_CPW\_ERR\_RECORD\_LOCKED\_BY\_ANOTHER\_SESSION
- ADX\_CPW\_ERR\_INVALID\_BUFFER
- ADX\_CPW\_ERR\_INVALID\_DATA
- ADX\_CPW\_ERR\_UNSUPPORTED\_FOR\_MODELS

## ValidateID

ValidateID determines whether a user ID exists in the enhanced authorization file and return “successful” if it does. ValidateID does not require a session ID.

If the given ID is a model ID, then ValidateID indicates that the ID was not found (ADX\_CPW\_ERR\_ID\_NOT\_FOUND).

### C interface:

```
long ADX_CPW_ValidateID(const char *id);
```

### CBASIC interface:

```
SUB Adx.Cpw.ValidateID(rc, id) EXTERNAL
    INTEGER*4 rc
    STRING id
END SUB
```

### Java interface:

```
(com.ibm.OS4690.security.EnhancedSecurity)
boolean isValidId(String id);
```

### Return codes:

- 0 — User ID is valid.

ADX\_CPW\_ERR\_SERVICE\_NOT\_AVAILABLE  
ADX\_CPW\_ERR\_ID\_NOT\_FOUND  
ADX\_CPW\_ERR\_RECORD\_LOCKED\_BY\_ANOTHER\_SESSION  
ADX\_CPW\_ERR\_INVALID\_BUFFER

## ChangePassword

ChangePassword changes the password of the specified user ID (*id*). The current password must be provided in the *oldpw* parameter and the new password in the *newpw* parameter. ChangePassword does not require a session id. The password cannot be changed for model records.

### C interface:

```
long ADX_CPW_ChangePassword(const char *id, const char *oldpw, const char *newpw);
```

### CBASIC interface:

```
SUB Adx.Cpw.ChangePassword(rc, id, oldpw, newpw) EXTERNAL
    INTEGER*4 rc
    STRING id
    STRING oldpw
    STRING newpw
END SUB
```

### Java interface:

```
(com.ibm.OS4690.security.EnhancedSecurity)
void changePassword(String id, String oldPassword, String newPassword);
```

### Return codes:

0 — Update successful  
ADX\_CPW\_ERR\_SERVICE\_NOT\_AVAILABLE  
ADX\_CPW\_ERR\_ID\_NOT\_FOUND  
ADX\_CPW\_ERR\_PASSWORD\_DOES\_NOT\_MATCH  
ADX\_CPW\_ERR\_PASSWORD\_SAME\_AS\_PREVIOUS  
ADX\_CPW\_ERR\_PASSWORD\_CONTAINS\_ID  
ADX\_CPW\_ERR\_PASSWORD\_CONTAINS\_SEQUENCE  
ADX\_CPW\_ERR\_PASSWORD\_CONTAINS\_REPETITION  
ADX\_CPW\_ERR\_TOO\_FEW\_CHARACTERS\_CHANGED  
ADX\_CPW\_ERR\_PASSWORD\_MUST\_CONTAIN\_CHAR\_OF\_EACH\_TYPE  
ADX\_CPW\_ERR\_PASSWORD\_MUST\_CONTAIN\_ALPHA\_AND\_NUMERIC  
ADX\_CPW\_ERR\_RECORD\_LOCKED\_BY\_ANOTHER\_SESSION  
ADX\_CPW\_ERR\_INVALID\_BUFFER  
ADX\_CPW\_ERR\_INVALID\_DATA  
ADX\_CPW\_ERR\_INVALID\_PASSWORD  
ADX\_CPW\_ERR\_PASSWORD\_TOO\_SHORT  
ADX\_CPW\_ERR\_UNSUPPORTED\_FOR\_MODELS

## GetAllAttributes

GetAllAttributes returns the authorization attributes and user-defined attributes for the provided user ID (*id*). The correct password (*pw*) for the user is required. No session ID is required. The attributes are returned in the location pointed to by the *buffer* parameter.

The complete string is 118 characters. It is generated by appending the 8 character user-defined attribute string to the end of the 110 character authorization attribute string. See “Authorization attributes” on page 680 and “User-defined attributes” on page 681 for more details.

In C, a partial string of attributes can be requested by providing a buffer size (*buffsize*) less than 118 bytes long. In CBASIC and Java, the entire attribute string is always returned.

#### C interface:

```
long ADX_CPW_GetAllAttributes(const char *id, const char *pw,
                             char *buffer, long buffsize);
```

#### CBASIC interface:

```
SUB Adx.Cpw.GetAllAttributes(rc, id, pw, buffer) EXTERNAL
    INTEGER*4 rc
    STRING id
    STRING pw
    STRING buffer
END SUB
```

#### Java interface:

```
(com.ibm.OS4690.security.EnhancedSecurity)
String getAllAttributes(String id, String password);
```

#### Return codes:

- 0 — Success
- ADX\_CPW\_ERR\_SERVICE\_NOT\_AVAILABLE
- ADX\_CPW\_ERR\_ID\_NOT\_FOUND
- ADX\_CPW\_ERR\_PASSWORD\_DOES\_NOT\_MATCH
- ADX\_CPW\_ERR\_PASSWORD\_EXPIRED
- ADX\_CPW\_ERR\_INVALID\_DATA
- ADX\_CPW\_ERR\_INVALID\_BUFFER
- ADX\_CPW\_ERR\_RECORD\_LOCKED\_BY\_ANOTHER\_SESSION

## GetAllAttributesCurrentUser

GetAllAttributesCurrentUser returns the authorization attributes and user-defined attributes for the currently logged in user. The attributes are returned in the location pointed to by the *buffer* parameter. If the user ID has been deleted or the password has expired since the user logged in or the user record is locked, this call fails with an appropriate error.

The user ID for a process that was not started by a specific user’s login session (or a background process) is considered to be “BACKGRND”. If this API is called from a background process and the ID “BACKGRND” does not exist, the error code ADX\_CPW\_ERR\_ID\_NOT\_FOUND is returned. Otherwise, this API functions as it would for a logged in user ID.

The complete string is 118 characters. It is generated by appending the 8 character user-defined attribute string to the end of the 110 character authorization attribute string. See “Authorization attributes” on page 680 and “User-defined attributes” on page 681 for more details.

In C, a partial string of attributes can be requested by providing a buffer size (*buffsize*) less than 118 bytes long. In CBASIC and Java, the entire attribute string is always returned.

#### C interface:

```
long ADX_CPW_GetAllAttributesCurrentUser(char *buffer, long buffsize);
```

**CBASIC interface:**

```
SUB Adx.Cpw.GetAllAttributesCurrentUser(rc, buffer) EXTERNAL
    INTEGER*4 rc
    STRING buffer
END SUB
```

**Java interface:**

```
(com.ibm.OS4690.security.EnhancedSecurity)
String getAllAttributesCurrentUser();
```

**Return codes:**

```
0 — Success
ADX_CPW_ERR_SERVICE_NOT_AVAILABLE
ADX_CPW_ERR_ID_NOT_FOUND
ADX_CPW_ERR_INVALID_BUFFER
ADX_CPW_ERR_PASSWORD_EXPIRED
ADX_CPW_ERR_RECORD_LOCKED_BY_ANOTHER_SESSION
```

**StartSession**

StartSession starts a session with the enhanced password management system. A session is required to get and modify most authorization data. Users not authorized to modify security settings are not allowed to start a session. The user ID (*id*) and password (*pw*) must be provided.

If successful, StartSession returns 0 and places the session number in the location referred to by *snum*.

**C interface:**

```
long ADX_CPW_StartSession(const char *id, const char *pw, long *snum);
```

**CBASIC interface:**

```
SUB Adx.Cpw.StartSession(rc, id, pw, snum) EXTERNAL
    INTEGER*4 rc
    STRING id
    STRING pw
    INTEGER*4 snum
END SUB
```

**Java interface:**

```
(com.ibm.OS4690.security.EnhancedSecurity)
EnhancedSecuritySession startSession(String id, String password);
```

**Return codes:**

- 0 — Success
- ADX\_CPW\_ERR\_SERVICE\_NOT\_AVAILABLE
- ADX\_CPW\_ERR\_ID\_NOT\_FOUND
- ADX\_CPW\_ERR\_ID\_NOT\_AUTHORIZED
- ADX\_CPW\_ERR\_PASSWORD\_DOES\_NOT\_MATCH
- ADX\_CPW\_ERR\_PASSWORD\_EXPIRED
- ADX\_CPW\_ERR\_SESSION\_LIMIT\_EXCEEDED
- ADX\_CPW\_ERR\_INVALID\_BUFFER
- ADX\_CPW\_ERR\_INVALID\_DATA
- ADX\_CPW\_ERR\_NOT\_ACTING\_MASTER

## StartSessionCurrentUser

StartSessionCurrentUser starts a session using the authority given to the currently logged in user. A session is required to get and modify most authorization data. Users not authorized to modify security settings will not be allowed to start a session. If the user ID has been deleted or the password has expired since the user logged in or if the user record is locked, this call fails with an appropriate error.

The user ID for a process that was not started by a specific user's login session (or a background process) is considered to be "BACKGRND". If this API is called from a background process and the ID "BACKGRND" does not exist, the error code ADX\_CPW\_ERR\_ID\_NOT\_FOUND is returned. Otherwise, this API functions as it would for a logged in user ID.

If successful, StartSessionCurrentUser returns 0 and places the session number in the location referred to by *snum*.

### C interface:

```
long ADX_CPW_StartSessionCurrentUser(long *snum);
```

### CBASIC interface:

```
SUB Adx.Cpw.StartSessionCurrentUser(rc, snum) EXTERNAL
    INTEGER*4 rc
    INTEGER*4 snum
END SUB
```

### Java interface:

```
(com.ibm.OS4690.security.EnhancedSecurity)
EnhancedSecuritySession startSessionCurrentUser();
```

### Return codes:

- 0 — Success
- ADX\_CPW\_ERR\_SERVICE\_NOT\_AVAILABLE
- ADX\_CPW\_ERR\_ID\_NOT\_FOUND
- ADX\_CPW\_ERR\_ID\_NOT\_AUTHORIZED
- ADX\_CPW\_ERR\_SESSION\_LIMIT\_EXCEEDED
- ADX\_CPW\_ERR\_PASSWORD\_EXPIRED
- ADX\_CPW\_ERR\_NOT\_ACTING\_MASTER

## GetAttributes

GetAttributes returns the authorization attributes for the provided user ID (*id*) in the location indicated by the *buffer* parameter.

The complete attribute string is 110 bytes and consists of the 104 individual attributes, the 3 character group number, and the 3 character user number. The user-defined attributes are not returned.

In C, a partial string of attributes can be requested by providing a buffer size (*buffsize*) of less than 110 bytes long. In CBASIC and Java, the entire attribute string is always returned.

### C interface:

```
long ADX_CPW_GetAttributes(long snum, const char *id, char *buffer, long buffsize);
```

### CBASIC interface:



```

SUB Adx.Cpw.GetAttributes(rc, snum, id, buffer) EXTERNAL
    INTEGER*4 rc
    INTEGER*4 snum
    STRING id
    STRING buffer
END SUB

```

#### Java interface:

```

(com.ibm.OS4690.security.AuthorizationRecord)
String getAttributes();

```

#### Return codes:

```

0 — Success
ADX_CPW_ERR_ID_NOT_FOUND
ADX_CPW_ERR_RECORD_LOCKED_BY_ANOTHER_SESSION
ADX_CPW_ERR_SESSION_TIME_OUT
ADX_CPW_ERR_INVALID_SESSION_ID
ADX_CPW_ERR_NOT_ACTING_MASTER
ADX_CPW_ERR_INVALID_BUFFER
ADX_CPW_ERR_INVALID_DATA

```

## GetAttribute

GetAttribute returns a single authorization attribute for the provided operator ID (*id*). The attribute index should be specified by the *index* parameter. The result is put into the first byte indicated by the *buffer* parameter. You can only get one of the 104 individual attribute values; you cannot get the user number or the group number with GetAttribute.

#### C interface:

```

long Adx_Cpw_GetAttribute(long snum, const char *id, short index, char *buffer);

```

#### CBASIC interface:

```

SUB Adx.Cpw.GetAttribute(rc, snum, id, index, buffer) EXTERNAL
    INTEGER*4 rc
    INTEGER*4 snum
    STRING id
    INTEGER*2 index
    STRING buffer
END SUB

```

#### Java interface:

```

(com.ibm.OS4690.security.AuthorizationRecord)
boolean getAttribute(int index);

```

#### Return codes:

```

0 — Success
ADX_CPW_ERR_ID_NOT_FOUND
ADX_CPW_ERR_RECORD_LOCKED_BY_ANOTHER_SESSION
ADX_CPW_ERR_SESSION_TIME_OUT
ADX_CPW_ERR_INVALID_SESSION_ID
ADX_CPW_ERR_NOT_ACTING_MASTER
ADX_CPW_ERR_INVALID_BUFFER
ADX_CPW_ERR_INVALID_DATA

```

## GetUserAttributes

GetUserAttributes returns the user-defined attributes for the provided operator ID (*id*) in the location indicated by the *buffer* parameter.

The complete attribute string is 8 long. In C, a partial string of attributes can be requested by providing a buffer size (*buffsize*) of less than 8 bytes. In CBASIC and Java, the entire attribute string is always returned.

The Java API has a convenience function that can be used to get a single user attribute (similar to the GetAttribute API).

### C interface:

```
long ADX_CPW_GetUserAttributes(long snum, const char *id,
                               char *buffer, long buffsize);
```

### CBASIC interface:

```
SUB Adx.Cpw.GetUserAttributes(rc, snum, id, buffer) EXTERNAL
    INTEGER*4 rc
    INTEGER*4 snum
    STRING id
    STRING buffer
END SUB
```

### Java interface:

```
(com.ibm.OS4690.security.AuthorizationRecord)
String getUserAttributes();
boolean getUserAttribute(int index);
```

### Return codes:

```
0 — Success
ADX_CPW_ERR_RECORD_LOCKED_BY_ANOTHER_SESSION
ADX_CPW_ERR_ID_NOT_FOUND
ADX_CPW_ERR_SESSION_TIME_OUT
ADX_CPW_ERR_INVALID_SESSION_ID
ADX_CPW_ERR_NOT_ACTING_MASTER
ADX_CPW_ERR_INVALID_BUFFER
ADX_CPW_ERR_INVALID_DATA
```

## Lock

Locks the user record with the indicated ID (*id*). This allows you to update the record and prevents other sessions accessing the record. Sessions cannot write records until they are locked.

Lock returns 80E70021 when the user attempts to lock a record that has an authorization level greater than his own.

### C interface:

```
long ADX_CPW_Lock(long snum, char *id);
```

### CBASIC interface:

```
SUB Adx.Cpw.Lock(rc, snum, id) EXTERNAL
    INTEGER*4 rc
    INTEGER*4 snum
    STRING id
END SUB
```

**Java interface:**

```
(com.ibm.OS4690.security.AuthorizationRecord)
void lock();
```

**Return codes:**

```
0 — Success
0x80E70001 — ADX_CPW_ERR_ID_NOT_FOUND
0x80E7000D — ADX_CPW_ERR_RECORD_LOCKED_BY_ANOTHER_SESSION
0x80E7000E — ADX_CPW_ERR_SESSION_TIME_OUT
0x80E7000F — ADX_CPW_ERR_INVALID_BUFFER
0x80E70011 — ADX_CPW_ERR_INVALID_SESSION_ID
0x80E70012 — ADX_CPW_ERR_INVALID_DATA
0x80E70018 — ADX_CPW_ERR_NOT_ACTING_MASTER
0x80E70021 — ADX_CPW_ERR_AUTHORIZATION_LEVEL_ERROR
```

**Copy**

Copy creates a new authorization record by copying the attributes from the user specified by the sourceid variable. The new record is automatically locked when it is created. The record is created with the specified ID (targetid) and password (targetpw). Lowercase characters in the user ID are converted to uppercase before being used.

The flag parameter indicates the behavior of this function. The valid values are as follows:

**Note:** These are character digits (not binary values).

```
0 — Create a user record with a non-expired password.
1 — Create a user record with an expired password.
2 — New record is a model. In this case the password is not used, but must still be set to a valid
pointer (an empty string is acceptable).
```

Copy returns a 0x80E70021 error if the source ID has an authorization level greater than the current user's. If copy is successful, the new record's authorization attributes are the intersection of the model record's attributes and the user's attributes. Attribute A will be enabled in the new record only if attribute A is enabled in the user's record AND in the model's record.

**Note:** A return code of 0 (success) means the record was created successfully and does not imply anything about the state of the new record's authorization flags.

**C interface:**

```
long ADX_CPW_Copy(long snum, char flags, char *sourceid,
                  char *targetid, char *targetpw);
```

**CBASIC interface:**

```
SUB Adx.Cpw.Copy(rc, snum, flags, sourceid, targetid, targetpw) EXTERNAL
    INTEGER* rc
    INTEGER*4 snum
    STRING flags
    STRING sourceid
    STRING targetid
    STRING targetpw
END SUB
```

**Java interface:**

```
(com.ibm.OS4690.security.EnhancedSecuritySession)
AuthorizationRecord copyRecord(String sourceId, String targetId,
String targetPassword, boolean expirePassword, boolean createModelRecord);
```

#### Return codes:

0 — Success

0x80E70001 — ADX\_CPW\_ERR\_ID\_NOT\_FOUND

0x80E70006 — ADX\_CPW\_ERR\_ID\_ALREADY\_EXISTS

0x80E70008 — ADX\_CPW\_ERR\_PASSWORD\_CONTAINS\_ID

0x80E70009 — ADX\_CPW\_ERR\_PASSWORD\_CONTAINS\_SEQUENCE

0x80E7000A — ADX\_CPW\_ERR\_PASSWORD\_CONTAINS\_REPETITION

0x80E7000C — ADX\_CPW\_ERR\_PASSWORD\_MUST\_CONTAIN\_CHAR\_OF\_EACH\_TYPE

0x80E7000D — ADX\_CPW\_ERR\_RECORD\_LOCKED\_BY\_ANOTHER\_SESSION

0x80E7000E — ADX\_CPW\_ERR\_SESSION\_TIME\_OUT

0x80E7000F — ADX\_CPW\_ERR\_INVALID\_BUFFER

0x80E70011 — ADX\_CPW\_ERR\_INVALID\_SESSION\_ID

0x80E70012 — ADX\_CPW\_ERR\_INVALID\_DATA

0x80E70014 — ADX\_CPW\_ERR\_INVALID\_TARGET\_ID

0x80E70015 — ADX\_CPW\_ERR\_INVALID\_PASSWORD

0x80E70016 — ADX\_CPW\_ERR\_PASSWORD\_TOO\_SHORT

0x80E70018 — ADX\_CPW\_ERR\_NOT\_ACTING\_MASTER

0x80E70019 — ADX\_CPW\_ERR\_PASSWORD\_FILE\_FULL

0x80E7001E — ADX\_CPW\_ERR\_PASSWORD\_MUST\_CONTAIN\_ALPHA\_AND\_NUMERIC

0x80E70021 — ADX\_CPW\_ERR\_AUTHORIZATION\_LEVEL\_ERROR

## SetAttributes

SetAttributes updates attributes for the authorization record indicated by the id parameter. The attributes are retrieved from the location indicated by the buffer parameter.

The complete attribute string can be up to 110 bytes long and consists of the 104 individual attributes, the 3 character group number, and the 3 character user number. The user-defined attributes are not included in the string.

In C, a partial string of attributes can be requested by providing a buffer size (buffsize) of less than 104 bytes long. In CBASIC and Java, the length of the string is used to determine the number of attributes to update.

An individual attribute value in the string must be set to either a valid value (as described in “Authorization attributes” on page 680 and “User-defined attributes” on page 681) or to the appropriate number of blanks (‘ ’).

SetAttributes returns an 80E70020 error if the user tries to enable an attribute that is disabled in their own record.

SetAttributes has always returned an 80E7001D error when an invalid set of attributes is specified. For example, setting Enhanced Security to disabled and Password Settings to enabled will cause this error. 80E7001D is returned before the check for 80E70020 in cases where both errors are present.

#### C interface:

```
long ADX_CPW_SetAttributes(long snum, char *id, char *buffer, long buffsize);
```

**CBASIC interface:**

```
SUB Adx.Cpw.SetAttributes(rc, snum, id, buffer) EXTERNAL
    INTEGER*4 rc
    INTEGER*4 snum
    STRING id
    STRING buffer
END SUB
```

**Java interface:**

```
(com.ibm.OS4690.security.AuthorizationRecord)
void setAttributes(String attributes);
```

**Return codes:**

- 0 — Success
- 0x80E70001 — ADX\_CPW\_ERR\_ID\_NOT\_FOUND
- 0x80E7000D — ADX\_CPW\_ERR\_RECORD\_LOCKED\_BY\_ANOTHER\_SESSION
- 0x80E7000E — ADX\_CPW\_ERR\_SESSION\_TIME\_OUT
- 0x80E7000F — ADX\_CPW\_ERR\_INVALID\_BUFFER
- 0x80E70011 — ADX\_CPW\_ERR\_INVALID\_SESSION\_ID
- 0x80E70012 — ADX\_CPW\_ERR\_INVALID\_DATA
- 0x80E70013 — ADX\_CPW\_ERR\_RECORD\_NOT\_LOCKED\_BY\_THIS\_SESSION
- 0x80E70018 — ADX\_CPW\_ERR\_NOT\_ACTING\_MASTER
- 0x80E7001D — ADX\_CPW\_ERR\_ATTRIBUTE\_MISMATCH
- 0x80E70020 — ADX\_CPW\_ERR\_UNAUTHORIZED\_ATTRIBUTE\_SET

**SetAttribute**

SetAttribute sets a single authorization attribute for the provided operator ID (id). The attribute index should be specified by the index parameter. The attribute value should be specified by the value parameter. You can only set one of the 104 individual attribute values; you cannot set the user number or the group number with this function.

SetAttribute returns an 80E70020 error if the user tries to enable an attribute that is disabled in his own record.

SetAttribute has always returned an 80E7001D error when an invalid attribute is specified. For example, setting Enhanced Security to disabled when Password Settings is enabled will cause this error. 80E7001D is returned before the check for 80E70020 in cases where both errors are present.

**C interface:**

```
long ADX_CPW_SetAttribute(long snum, const char *id, short index, char attr);
```

**CBASIC interface:**

```
SUB Adx.Cpw.SetAttribute(rc, snum, id, index, value) EXTERNAL
    INTEGER*4 rc
    INTEGER*4 snum
    STRING id
    INTEGER*2 index
    STRING value
END SUB
```

**Java interface:**

```
(com.ibm.OS4690.security.AuthorizationRecord)
void setAttribute(int index, boolean value);
```

**Return codes:**

0 — Success  
 0x80E70001 — ADX\_CPW\_ERR\_ID\_NOT\_FOUND  
 0x80E7000D — ADX\_CPW\_ERR\_RECORD\_LOCKED\_BY\_ANOTHER\_SESSION  
 0x80E7000E — ADX\_CPW\_ERR\_SESSION\_TIME\_OUT  
 0x80E7000F — ADX\_CPW\_ERR\_INVALID\_BUFFER  
 0x80E70011 — ADX\_CPW\_ERR\_INVALID\_SESSION\_ID  
 0x80E70012 — ADX\_CPW\_ERR\_INVALID\_DATA  
 0x80E70013 — ADX\_CPW\_ERR\_RECORD\_NOT\_LOCKED\_BY\_THIS\_SESSION  
 0x80E70018 — ADX\_CPW\_ERR\_NOT\_ACTING\_MASTER  
 0x80E7001D — ADX\_CPW\_ERR\_ATTRIBUTE\_MISMATCH  
 0x80E70020 — ADX\_CPW\_ERR\_UNAUTHORIZED\_ATTRIBUTE\_SET

**SetUserAttributes**

SetUserAttributes updates the user attributes for the record indicated by the *id* parameter. The attributes are retrieved from the location indicated by the *buffer* parameter.

The complete attribute string can be up to 8 bytes long. In C, a partial string of attributes can be requested by providing a buffer size (*buffsize*) of less than 8 bytes. In CBASIC and Java, the length of the string is used to determine the number of attributes to update.

The Java API has a convenience function that can be used to set a single user attribute (similar to the SetAttribute API).

SetUserAttributes returns an 80E70020 error if the user tries to enable an attribute that is disabled in their own record.

**C interface:**

```
long ADX_CPW_SetUserAttributes(long snum, const char *id,
                               const char *buffer, long buffsize);
```

**CBASIC interface:**

```
SUB Adx.Cpw.SetUserAttributes(rc, snum, id, buffer) EXTERNAL
  INTEGER*4 rc
  INTEGER*4 snum
  STRING id
  STRING buffer
END SUB
```

**Java interface:**

```
(com.ibm.OS4690.security.AuthorizationRecord)
void setUserAttributes(String attributes);
void setUserAttribute(int index, boolean value);
```

**Return codes:**

0 — Success  
 0x80E70001 — ADX\_CPW\_ERR\_ID\_NOT\_FOUND  
 0x80E7000D — ADX\_CPW\_ERR\_RECORD\_LOCKED\_BY\_ANOTHER\_SESSION  
 0x80E7000E — ADX\_CPW\_ERR\_SESSION\_TIME\_OUT  
 0x80E7000F — ADX\_CPW\_ERR\_INVALID\_BUFFER  
 0x80E70011 — ADX\_CPW\_ERR\_INVALID\_SESSION\_ID  
 0x80E70012 — ADX\_CPW\_ERR\_INVALID\_DATA

0x80E70013 — ADX\_CPW\_ERR\_RECORD\_NOT\_LOCKED\_BY\_THIS\_SESSION  
0x80E70018 — ADX\_CPW\_ERR\_NOT\_ACTING\_MASTER  
0x80E70020 — ADX\_CPW\_ERR\_UNAUTHORIZED\_ATTRIBUTE\_SET

## GetGroupNumber

GetGroupNumber retrieves the group number of the given user (*id*) and places it into the location referred to by the *groupnum* variable.

### C interface:

```
long ADX_CPW_GetGroupNumber(long snum, const char *id, short *groupnum);
```

### CBASIC interface:

```
SUB Adx.Cpw.GetGroupNumber(rc, snum, id, groupnum) EXTERNAL
    INTEGER*4 rc
    INTEGER*4 snum
    STRING id
    INTEGER*2 groupnum
END SUB
```

### Java interface:

```
(com.ibm.OS4690.security.AuthorizationRecord)
short getGroupNumber();
```

### Return codes:

0 — Success  
ADX\_CPW\_ERR\_ID\_NOT\_FOUND  
ADX\_CPW\_ERR\_RECORD\_LOCKED\_BY\_ANOTHER\_SESSION  
ADX\_CPW\_ERR\_SESSION\_TIME\_OUT  
ADX\_CPW\_ERR\_INVALID\_SESSION\_ID  
ADX\_CPW\_ERR\_NOT\_ACTING\_MASTER  
ADX\_CPW\_ERR\_INVALID\_BUFFER  
ADX\_CPW\_ERR\_INVALID\_DATA

## SetGroupNumber

SetGroupNumber sets the group number for the specified user ID (*id*) using the value in the *groupnum* parameter. The value can range from 2 to 254.

SetGroupNumber returns an 80E70020 error if the user tries to set a group number that is lower than their own group number.

### C interface:

```
long ADX_CPW_SetGroupNumber(long snum, const char *id, short groupnum);
```

### CBASIC interface:

```
SUB Adx.Cpw.SetGroupNumber(rc, snum, id, groupnum) EXTERNAL
    INTEGER*4 rc
    INTEGER*4 snum
    STRING id
    INTEGER*2 groupnum
END SUB
```

### Java interface:

```
(com.ibm.OS4690.security.AuthorizationRecord)
void setGroupNumber(short groupNumber);
```

**Return codes:**

0 — Success  
0x80E70001 — ADX\_CPW\_ERR\_ID\_NOT\_FOUND  
0x80E7000D — ADX\_CPW\_ERR\_RECORD\_LOCKED\_BY\_ANOTHER\_SESSION  
0x80E7000E — ADX\_CPW\_ERR\_SESSION\_TIME\_OUT  
0x80E7000F — ADX\_CPW\_ERR\_INVALID\_BUFFER  
0x80E70011 — ADX\_CPW\_ERR\_INVALID\_SESSION\_ID  
0x80E70012 — ADX\_CPW\_ERR\_INVALID\_DATA  
0x80E70013 — ADX\_CPW\_ERR\_RECORD\_NOT\_LOCKED\_BY\_THIS\_SESSION  
0x80E70018 — ADX\_CPW\_ERR\_NOT\_ACTING\_MASTER  
0x80E70020 — ADX\_CPW\_ERR\_UNAUTHORIZED\_ATTRIBUTE\_SET

## GetUserNumber

GetUserNumber retrieves the user number of the given user (*id*) and places it into the location referred to by the *usernum* variable.

**C interface:**

```
long ADX_CPW_GetUserNumber(long snum, const char *id, short *usernum);
```

**CBASIC interface:**

```
SUB Adx.Cpw.GetUserNumber(rc, snum, id, usernum) EXTERNAL
    INTEGER*4 rc
    INTEGER*4 snum
    STRING id
    INTEGER*2 usernum
END SUB
```

**Java interface:**

```
(com.ibm.OS4690.security.AuthorizationRecord) short getUserNumber();
```

**Return codes:**

0 — Success  
ADX\_CPW\_ERR\_ID\_NOT\_FOUND  
ADX\_CPW\_ERR\_RECORD\_LOCKED\_BY\_ANOTHER\_SESSION  
ADX\_CPW\_ERR\_SESSION\_TIME\_OUT  
ADX\_CPW\_ERR\_INVALID\_SESSION\_ID  
ADX\_CPW\_ERR\_NOT\_ACTING\_MASTER  
ADX\_CPW\_ERR\_INVALID\_BUFFER  
ADX\_CPW\_ERR\_INVALID\_DATA

## SetUserNumber

SetUserNumber updates the user number attribute for the specified authorization record (*id*) using the value in the *usernum* variable. The value can range from 1-254.

SetUserNumber will be updated to return an 80E70020 error if the user tries to set a user number that is lower than their own user number.

**C interface:**



```
long ADX_CPW_SetUserNumber(long snum, const char *id, short usernum);
```

#### **CBASIC interface:**

```
SUB Adx.Cpw.SetUserNumber(rc, snum, id, usernum) EXTERNAL
    INTEGER*4 rc
    INTEGER*4 snum
    STRING id
    INTEGER*2 usernum
END SUB
```

#### **Java interface:**

```
(com.ibm.OS4690.security.AuthorizationRecord)
void setUserNumber(short userNumber);
```

#### **Return codes:**

- 0 — Success
- 0x80E70001 — ADX\_CPW\_ERR\_ID\_NOT\_FOUND
- 0x80E7000D — ADX\_CPW\_ERR\_RECORD\_LOCKED\_BY\_ANOTHER\_SESSION
- 0x80E7000E — ADX\_CPW\_ERR\_SESSION\_TIME\_OUT
- 0x80E7000F — ADX\_CPW\_ERR\_INVALID\_BUFFER
- 0x80E70011 — ADX\_CPW\_ERR\_INVALID\_SESSION\_ID
- 0x80E70012 — ADX\_CPW\_ERR\_INVALID\_DATA
- 0x80E70013 — ADX\_CPW\_ERR\_RECORD\_NOT\_LOCKED\_BY\_THIS\_SESSION
- 0x80E70018 — ADX\_CPW\_ERR\_NOT\_ACTING\_MASTER
- 0x80E70020 — ADX\_CPW\_ERR\_UNAUTHORIZED\_ATTRIBUTE\_SET

## **GetAuthorizationLevel**

Gets the authorization level and returns it as a short.

#### **C interface:**

```
long ADX_CPW_GetAuthorizationLevel(long snum, char *id, short *level);
```

#### **CBASIC interface:**

```
SUB Adx.Cpw.GetAuthorizationLevel(rc, snum, id, level) EXTERNAL
    INTEGER*4 rc
    INTEGER*4 snum
    STRING id
    INTEGER*2 level
END SUB
```

#### **Java interface:**

```
(com.ibm.OS4690.security.AuthorizationRecord) short getAuthorizationLevel();
```

#### **Return codes:**

- 0 — Success
- 0x80E70001 — ADX\_CPW\_ERR\_ID\_NOT\_FOUND
- 0x80E7000D — ADX\_CPW\_ERR\_RECORD\_LOCKED\_BY\_ANOTHER\_SESSION
- 0x80E7000E — ADX\_CPW\_ERR\_SESSION\_TIME\_OUT
- 0x80E7000F — ADX\_CPW\_ERR\_INVALID\_BUFFER
- 0x80E70011 — ADX\_CPW\_ERR\_INVALID\_SESSION\_ID
- 0x80E70012 — ADX\_CPW\_ERR\_INVALID\_DATA
- 0x80E70018 — ADX\_CPW\_ERR\_NOT\_ACTING\_MASTER

## SetAuthorizationLevel

Updates the authorization level for the specified authorization record.

80E70021 is returned when a user successfully locked a record, but then tried to set the authorization level greater than his user level.

### C interface:

```
long ADX_CPW_SetAuthorizationLevel(long snum, char *id, short level);
```

### CBASIC interface:

```
SUB Adx.Cpw.SetAuthorizationLevel(rc, snum, id, level) EXTERNAL
    INTEGER*4 rc
    INTEGER*4 snum
    STRING id
    INTEGER*2 level
END SUB
```

### Java interface:

```
(com.ibm.OS4690.security.AuthorizationRecord)
void setAuthorizationLevel(short authorizationLevel);
```

### Return codes:

- 0 — Success
- 0x80E70001 — ADX\_CPW\_ERR\_ID\_NOT\_FOUND
- 0x80E7000D — ADX\_CPW\_ERR\_RECORD\_LOCKED\_BY\_ANOTHER\_SESSION
- 0x80E7000E — ADX\_CPW\_ERR\_SESSION\_TIME\_OUT
- 0x80E70011 — ADX\_CPW\_ERR\_INVALID\_SESSION\_ID
- 0x80E70012 — ADX\_CPW\_ERR\_INVALID\_DATA
- 0x80E70013 — ADX\_CPW\_ERR\_RECORD\_NOT\_LOCKED\_BY\_THIS\_SESSION
- 0x80E70018 — ADX\_CPW\_ERR\_NOT\_ACTING\_MASTER
- 0x80E70021 — ADX\_CPW\_ERR\_AUTHORIZATION\_LEVEL\_ERROR

## SetExpires

SetExpires sets the global setting for the number of days until a password expires to the value indicated by the expires parameter. The valid values for the expires parameter are 1, 7, 10, 14, 30, 45, 60, 90, 120, and 180.

SetExpires returns an 80E70022 error if the user is not authorized to modify Enhanced Security Password Settings.

### C interface:

```
long ADX_CPW_SetExpires(long snum, short expires);
```

### CBASIC interface:

```
SUB Adx.Cpw.SetExpires(rc, snum, expires) EXTERNAL
    INTEGER*4 rc
    INTEGER*4 snum
    INTEGER*2 expires
END SUB
```

### Java interface:

```
(com.ibm.OS4690.security.EnhancedSecurityPasswordSettings)
void setExpires(short expires);
```

**Return codes:**

- 0 — Success
- 0x80E7000E — ADX\_CPW\_ERR\_SESSION\_TIME\_OUT
- 0x80E7000F — ADX\_CPW\_ERR\_INVALID\_BUFFER
- 0x80E70011 — ADX\_CPW\_ERR\_INVALID\_SESSION\_ID
- 0x80E70012 — ADX\_CPW\_ERR\_INVALID\_DATA
- 0x80E70018 — ADX\_CPW\_ERR\_NOT\_ACTING\_MASTER
- 0x80E70022 — ADX\_CPW\_ERR\_PASSWORD\_SETTINGS\_AUTHORIZATION\_ERROR

**Create**

This function creates a new authorization record using the system default attributes. The new record is automatically locked when it is created. The record is created with the specified ID (*id*) and password (*pw*). Lowercase characters in the user ID are converted to uppercase before being used.

The *flags* parameter indicates the behavior of this function. The valid values are as follows:

**Note:** These are character digits (not binary values).

- 0 — Create a user record with a non-expired password.
- 1 — Create a user record with an expired password.
- 2 — Create a model record. In this case the password is not used, but must still be set to a valid pointer (an empty string is acceptable).

**C interface:**

```
long ADX_CPW_Create(long snum, char flags, const char *id, const char *pw);
```

**CBASIC interface:**

```
SUB Adx.Cpw.Create(rc, snum, flags, id, pw) EXTERNAL
    INTEGER*4 rc
    INTEGER*4 snum
    STRING    flags
    STRING    id
    STRING    pw
END SUB
```

**Java interface:**

```
(com.ibm.OS4690.security.EnhancedSecuritySession)
AuthorizationRecord createRecord(String id, String password,
boolean expirePassword, boolean createModelRecord);
```

**Return codes:**

- 0 — Success
- ADX\_CPW\_ERR\_ID\_ALREADY\_EXISTS
- ADX\_CPW\_ERR\_PASSWORD\_CONTAINS\_ID
- ADX\_CPW\_ERR\_PASSWORD\_CONTAINS\_SEQUENCE
- ADX\_CPW\_ERR\_PASSWORD\_CONTAINS\_REPETITION
- ADX\_CPW\_ERR\_PASSWORD\_MUST\_CONTAIN\_CHAR\_OF\_EACH\_TYPE
- ADX\_CPW\_ERR\_PASSWORD\_MUST\_CONTAIN\_ALPHA\_AND\_NUMERIC
- ADX\_CPW\_ERR\_RECORD\_LOCKED\_BY\_ANOTHER\_SESSION
- ADX\_CPW\_ERR\_SESSION\_TIME\_OUT
- ADX\_CPW\_ERR\_INVALID\_SESSION\_ID
- ADX\_CPW\_ERR\_NOT\_ACTING\_MASTER

```

ADX_CPW_ERR_INVALID_BUFFER
ADX_CPW_ERR_INVALID_DATA
ADX_CPW_ERR_INVALID_TARGET_ID
ADX_CPW_ERR_INVALID_PASSWORD
ADX_CPW_ERR_PASSWORD_TOO_SHORT
ADX_CPW_ERR_PASSWORD_FILE_FULL

```

## Rename

Rename changes the ID of a user (*id*) to a new value (*newid*). Lowercase characters in the user ID are converted to uppercase before being used. The record being renamed must be locked by this session.

### C interface:

```
long ADX_CPW_Rename(long snum, const char *id, const char *newid);
```

### CBASIC interface:

```

SUB Adx.Cpw.Rename(rc, snum, id, newid) EXTERNAL
    INTEGER*4 rc
    INTEGER*4 snum
    STRING id
    STRING newid
END SUB

```

### Java interface:

```

(com.ibm.OS4690.security.AuthorizationRecord)
void setId(String newid);

```

### Return codes:

```

0 — Success
ADX_CPW_ERR_ID_ALREADY_EXISTS
ADX_CPW_ERR_RECORD_LOCKED_BY_ANOTHER_SESSION
ADX_CPW_ERR_SESSION_TIME_OUT
ADX_CPW_ERR_INVALID_SESSION_ID
ADX_CPW_ERR_NOT_ACTING_MASTER
ADX_CPW_ERR_INVALID_BUFFER
ADX_CPW_ERR_INVALID_DATA
ADX_CPW_ERR_INVALID_TARGET_ID
ADX_CPW_ERR_RECORD_NOT_LOCKED_BY_THIS_SESSION

```

## Delete

This function deletes the authorization record indicated by the *id* parameter.

### C interface:

```
long ADX_CPW_Delete(long snum, const char *id);
```

### CBASIC interface:

```

SUB Adx.Cpw.Delete(rc, snum, id) EXTERNAL
    INTEGER*4 rc
    INTEGER*4 snum
    STRING id
END SUB

```

### Java interface:

```
(com.ibm.OS4690.security.EnhancedSecuritySession)
void delete(String id);
```

#### Return codes:

```
0 — Success
ADX_CPW_ERR_RECORD_LOCKED_BY_ANOTHER_SESSION
ADX_CPW_ERR_SESSION_TIME_OUT
ADX_CPW_ERR_INVALID_SESSION_ID
ADX_CPW_ERR_NOT_ACTING_MASTER
ADX_CPW_ERR_INVALID_BUFFER
ADX_CPW_ERR_INVALID_DATA
ADX_CPW_ERR_RECORD_NOT_LOCKED_BY_THIS_SESSION
ADX_CPW_ERR_UNSUPPORTED_FOR_MASTER_RECORD
```

## SetPassword

SetPassword sets the password for the authorization record indicated by *id* to the value *newpw*. The password cannot be changed for model records. Valid values for *flags* are:

```
0 — Set password as not expired.
1 — Set password as expired.
```

#### C interface:

```
long ADX_CPW_SetPassword(long snum, char flags, const char *id, const char *newpw);
```

#### CBASIC interface:

```
SUB Adx.Cpw.SetPassword(rc, snum, flags, id, newpw) EXTERNAL
    INTEGER*4 rc
    INTEGER*4 snum
    STRING flags
    STRING id
    STRING newpw
END SUB
```

#### Java interface:

```
(com.ibm.OS4690.security.AuthorizationRecord)
void setPassword(String password, boolean expirePassword);
```

#### Return codes:

```
0 — Success
ADX_CPW_ERR_PASSWORD_SAME_AS_PREVIOUS
ADX_CPW_ERR_PASSWORD_CONTAINS_ID
ADX_CPW_ERR_PASSWORD_CONTAINS_SEQUENCE
ADX_CPW_ERR_PASSWORD_CONTAINS_REPETITION
ADX_CPW_ERR_PASSWORD_MUST_CONTAIN_CHAR_OF_EACH_TYPE
ADX_CPW_ERR_PASSWORD_MUST_CONTAIN_ALPHA_AND_NUMERIC
ADX_CPW_ERR_RECORD_LOCKED_BY_ANOTHER_SESSION
ADX_CPW_ERR_SESSION_TIME_OUT
ADX_CPW_ERR_INVALID_SESSION_ID
ADX_CPW_ERR_NOT_ACTING_MASTER
ADX_CPW_ERR_INVALID_BUFFER
ADX_CPW_ERR_INVALID_DATA
```

```
ADX_CPW_ERR_RECORD_NOT_LOCKED_BY_THIS_SESSION
ADX_CPW_ERR_INVALID_PASSWORD
ADX_CPW_ERR_PASSWORD_TOO_SHORT
ADX_CPW_ERR_UNSUPPORTED_FOR_MODELS
```

## GetRecordCount

GetRecordCount returns the number of user authorization records defined in the file in the location indicated by the *count* parameter. The count includes both locked and unlocked user records, but not deleted or model records. The value returned is the number of IDs that would be returned by the GetIDs call.

### C interface:

```
long ADX_CPW_GetRecordCount(long snum, short *count);
```

### CBASIC interface:

```
SUB Adx.Cpw.GetRecordCount(rc, snum, rcount) EXTERNAL
    INTEGER*4 rc
    INTEGER*4 snum
    INTEGER*2 rcount
END SUB
```

### Java interface:

```
(com.ibm.OS4690.security.EnhancedSecuritySession)
int getRecordCount();
```

### Return codes:

```
0 — Success
ADX_CPW_ERR_SESSION_TIME_OUT
ADX_CPW_ERR_INVALID_SESSION_ID
ADX_CPW_ERR_NOT_ACTING_MASTER
ADX_CPW_ERR_INVALID_BUFFER
```

## GetIDs

GetIDs returns a string containing all of the IDs in the authorization file in the location indicated by the *buffer* variable. Each ID is nine characters, prefixed with blanks, if necessary, so that the string size divided by 16 equals the number of complete IDs returned.

If the buffer is too small to contain all of the IDs or is not a multiple of the ID size (16), a ADX\_CPW\_ERR\_INVALID\_BUFFER error is returned.

### C interface:

```
long ADX_CPW_GetIDs(long snum, char *buffer, long buffsize);
```

### CBASIC interface:

```
SUB Adx.Cpw.GetIDs(rc, snum, buffer) EXTERNAL
    INTEGER*4 rc
    INTEGER*4 snum
    STRING buffer
END SUB
```

### Java interface:

```
(com.ibm.OS4690.security.EnhancedSecuritySession)
String[] getIds();
```

**Return codes:**

0 — Success  
 ADX\_CPW\_ERR\_SESSION\_TIME\_OUT  
 ADX\_CPW\_ERR\_INVALID\_SESSION\_ID  
 ADX\_CPW\_ERR\_NOT\_ACTING\_MASTER  
 ADX\_CPW\_ERR\_INVALID\_BUFFER

**GetModelCount**

GetModelCount returns the number of authorization model records defined in the file in the location indicated by the *count* parameter. The count includes locked model records, but not deleted records. The value returned is the number of IDs that would be returned by the GetModelIDs call.

**C interface:**

```
long ADX_CPW_GetModelCount(long snum, short *count);
```

**CBASIC interface:**

```
SUB Adx.Cpw.GetModelCount(rc, snum, mcount) EXTERNAL
  INTEGER*4 rc
  INTEGER*4 snum
  INTEGER*2 mcount
END SUB
```

**Java interface:**

```
(com.ibm.OS4690.security.EnhancedSecuritySession)
int getModelRecordCount();
```

**Return codes:**

0 — Success  
 ADX\_CPW\_ERR\_SESSION\_TIME\_OUT  
 ADX\_CPW\_ERR\_INVALID\_SESSION\_ID  
 ADX\_CPW\_ERR\_NOT\_ACTING\_MASTER  
 ADX\_CPW\_ERR\_INVALID\_BUFFER

**GetModelIDs**

GetModelIDs returns a string containing all of the model IDs in the authorization file in the location indicated by the *buffer* variable. Each ID is nine characters, prefixed with blanks, if necessary, so that the string size divided by 16 equals the number of complete IDs returned.

If the buffer is too small to contain all of the IDs or is not a multiple of the ID size (16), a ADX\_CPW\_ERR\_INVALID\_BUFFER error is returned.

**C interface:**

```
long ADX_CPW_GetModelIDs(long snum, char *buffer, long bufsize);
```

**CBASIC interface:**

```
SUB Adx.Cpw.GetModelIDs(rc, snum, buffer) EXTERNAL
  INTEGER*4 rc
  INTEGER*4 snum
  STRING buffer
END SUB
```

**Java interface:**

```
(com.ibm.OS4690.security.EnhancedSecuritySession)
String[] getModelIds();
```

**Return codes:**

- 0 — Success
- ADX\_CPW\_ERR\_SESSION\_TIME\_OUT
- ADX\_CPW\_ERR\_INVALID\_SESSION\_ID
- ADX\_CPW\_ERR\_NOT\_ACTING\_MASTER
- ADX\_CPW\_ERR\_INVALID\_BUFFER

## GetExpires

GetExpires returns the global setting for the number of days until a password expires in the location indicated by the *expires* parameter.

**C interface:**

```
long ADX_CPW_GetExpires(long snum, short *expires);
```

**CBASIC interface:**

```
SUB Adx.Cpw.GetExpires(rc, snum, expires) EXTERNAL
    INTEGER*4 rc
    INTEGER*4 snum
    INTEGER*2 expires
END SUB
```

**Java interface:**

```
(com.ibm.OS4690.security.EnhancedSecurityPasswordSettings)
short getExpires();
```

**Return codes:**

- 0 — Success
- ADX\_CPW\_ERR\_SESSION\_TIME\_OUT
- ADX\_CPW\_ERR\_INVALID\_SESSION\_ID
- ADX\_CPW\_ERR\_NOT\_ACTING\_MASTER
- ADX\_CPW\_ERR\_INVALID\_BUFFER

## SetExpires

SetExpires sets the global setting for the number of days until a password expires to the value indicated by the *expires* parameter. The valid values for the *expires* parameter are 1, 7, 10, 14, 30, 45, 60, 90, 120, and 180.

**C interface:**

```
long ADX_CPW_SetExpires(long snum, short expires);
```

**CBASIC interface:**

```
SUB Adx.Cpw.SetExpires(rc, snum, expires) EXTERNAL
    INTEGER*4 rc
    INTEGER*4 snum
    INTEGER*2 expires
END SUB
```

**Java interface:**

```
(com.ibm.OS4690.security.EnhancedSecurityPasswordSettings)
void setExpires(short expires);
```



**Return codes:**

0 — Success  
 ADX\_CPW\_ERR\_SESSION\_TIME\_OUT  
 ADX\_CPW\_ERR\_INVALID\_SESSION\_ID  
 ADX\_CPW\_ERR\_NOT\_ACTING\_MASTER  
 ADX\_CPW\_ERR\_INVALID\_DATA

**GetMinPasswordLength**

GetMinPasswordLength returns the global setting for the minimum password length in the location indicated by the *length* parameter.

**C interface:**

```
long ADX_CPW_GetMinPasswordLength(long snum, short *length);
```

**CBASIC interface:**

```
SUB Adx.Cpw.GetMinPasswordLength (rc, snum, length) EXTERNAL
  INTEGER*4 rc
  INTEGER*4 snum
  INTEGER*2 length
END SUB
```

**Java interface:**

```
(com.ibm.OS4690.security.EnhancedSecurityPasswordSettings)
short getMinimumPasswordLength();
```

**Return codes:**

0 — Success  
 ADX\_CPW\_ERR\_SESSION\_TIME\_OUT  
 ADX\_CPW\_ERR\_INVALID\_SESSION\_ID  
 ADX\_CPW\_ERR\_NOT\_ACTING\_MASTER  
 ADX\_CPW\_ERR\_INVALID\_BUFFER

**SetMinPasswordLength**

SetMinPasswordLength sets the global setting for the minimum password length to the value indicated by the *newlength* parameter. The valid range for the minimum password length is normally 2 to 8. However, if the multiple character or minimum character change options are enabled, the valid range is 4 to 8.

SetMinPasswordLength returns an 80E70022 error if the user is not authorized to modify Enhanced Security Password Settings.

**C interface:**

```
long ADX_CPW_SetMinPasswordLength(long snum, short newlength);
```

**CBASIC interface:**

```
SUB Adx.Cpw.SetMinPasswordLength (rc, snum, newlength) EXTERNAL
  INTEGER*4 rc
  INTEGER*4 snum
  INTEGER*2 newlength
END SUB
```

**Java interface:**

```
(com.ibm.OS4690.security.EnhancedSecurityPasswordSettings)
void setMinimumPasswordLength(short minimumPasswordLength);
```

### Return codes:

- 0 — Success
- 0x80E7000E — ADX\_CPW\_ERR\_SESSION\_TIME\_OUT
- 0x80E7000F — ADX\_CPW\_ERR\_INVALID\_BUFFER
- 0x80E70011 — ADX\_CPW\_ERR\_INVALID\_SESSION\_ID
- 0x80E70012 — ADX\_CPW\_ERR\_INVALID\_DATA
- 0x80E7001C — ADX\_CPW\_ERR\_MIN\_PW\_LENGTH\_MISMATCH
- 0x80E70022 — ADX\_CPW\_ERR\_PASSWORD\_SETTINGS\_AUTHORIZATION\_ERROR

## GetExpireWarning

GetExpireWarning returns the global setting for the expiration warning time in the location indicated by the *warning* parameter. This setting corresponds to number of days that a password warning message appears before the ID's password expires.

### C interface:

```
long ADX_CPW_GetExpireWarning(long snum, short *warning);
```

### CBASIC interface:

```
SUB Adx.Cpw.GetExpireWarning(rc, snum, warning) EXTERNAL
    INTEGER*4 rc
    INTEGER*4 snum
    INTEGER*2 warning
END SUB
```

### Java interface:

```
(com.ibm.OS4690.security.EnhancedSecurityPasswordSettings)
short getExpireWarning();
```

### Return codes:

- 0 — Success
- ADX\_CPW\_ERR\_SESSION\_TIME\_OUT
- ADX\_CPW\_ERR\_INVALID\_SESSION\_ID
- ADX\_CPW\_ERR\_NOT\_ACTING\_MASTER
- ADX\_CPW\_ERR\_INVALID\_BUFFER

## SetExpireWarning

SetExpireWarning sets the global setting for the expiration warning time to the value indicated by the *newwarning* parameter. This setting corresponds to number of days that a password warning message appears before the ID's password expires. This value ranges from 1 to 15 (days).

SetExpireWarning returns an 80E70022 error if the user is not authorized to modify Enhanced Security Password Settings.

### C interface:

```
long ADX_CPW_SetExpireWarning(long snum, short newwarning);
```

### CBASIC interface:

```

SUB Adx.Cpw.SetExpireWarning(rc, snum, newwarning) EXTERNAL
    INTEGER*4 rc
    INTEGER*4 snum
    INTEGER*2 newwarning
END SUB

```

#### Java interface:

```

(com.ibm.OS4690.security.EnhancedSecurityPasswordSettings)
void setExpireWarning(short expireWarning);

```

#### Return codes:

```

0 — Success
0x80E7000E — ADX_CPW_ERR_SESSION_TIME_OUT
0x80E7000F — ADX_CPW_ERR_INVALID_BUFFER
0x80E70011 — ADX_CPW_ERR_INVALID_SESSION_ID
0x80E70012 — ADX_CPW_ERR_INVALID_DATA
0x80E70018 — ADX_CPW_ERR_NOT_ACTING_MASTER
0x80E70022 — ADX_CPW_ERR_PASSWORD_SETTINGS_AUTHORIZATION_ERROR

```

## GetMinChange

GetMinChange returns the global setting for the minimum character change option in the location indicated by the *minchange* parameter. The returned value will be **Y**, if the option is enabled, otherwise it will be **N**. When the option is enabled, new passwords must contain at least four characters that were not present in the previous password.

#### C interface:

```

long ADX_CPW_GetMinChange(long snum, char *minchange);

```

#### CBASIC interface:

```

SUB Adx.Cpw.GetMinChange(rc, snum, minchange) EXTERNAL
    INTEGER*4 rc
    INTEGER*4 snum
    STRING minchange
END SUB

```

#### Java interface:

```

(com.ibm.OS4690.security.EnhancedSecurityPasswordSettings)
boolean isMinimumCharacterChangeEnabled();

```

#### Return codes:

```

0 — Success
ADX_CPW_ERR_SESSION_TIME_OUT
ADX_CPW_ERR_INVALID_SESSION_ID
ADX_CPW_ERR_NOT_ACTING_MASTER
ADX_CPW_ERR_INVALID_BUFFER

```

## SetMinChange

SetMinChange sets the global setting for the minimum character change option to the value indicated by the *newminchange* parameter. When the option is enabled, new passwords must contain at least four characters that were not present in the previous password. Setting *newminchange* to **Y** enables the option, setting it to **N** disables it.

SetMinChange returns an 80E70022 error if the user is not authorized to modify Enhanced Security Password Settings.

**C interface:**

```
long ADX_CPW_SetMinChange(long snum, char newminchange);
```

**CBASIC interface:**

```
SUB Adx.Cpw.SetMinChange(rc, snum, newminchange) EXTERNAL
    INTEGER*4 rc
    INTEGER*4 snum
    STRING newminchange
END SUB
```

**Java interface:**

```
(com.ibm.OS4690.security.EnhancedSecurityPasswordSettings)
void setMinimumCharacterChangeEnabled(boolean minimumCharacterChangeEnabled);
```

**Return codes:**

- 0 — Success
- 0x80E7000E — ADX\_CPW\_ERR\_SESSION\_TIME\_OUT
- 0x80E7000F — ADX\_CPW\_ERR\_INVALID\_BUFFER
- 0x80E70011 — ADX\_CPW\_ERR\_INVALID\_SESSION\_ID
- 0x80E70012 — ADX\_CPW\_ERR\_INVALID\_DATA
- 0x80E70018 — ADX\_CPW\_ERR\_NOT\_ACTING\_MASTER
- 0x80E7001C — ADX\_CPW\_ERR\_MIN\_PW\_LENGTH\_MISMATCH
- 0x80E70022 — ADX\_CPW\_ERR\_PASSWORD\_SETTINGS\_AUTHORIZATION\_ERROR

## GetMultipleCharacter

GetMultipleCharacter returns the global setting for the multiple character option in the location indicated by the *multichar* parameter. The returned value will be **Y**, if the option is enabled; otherwise, it will be **N**.

**C interface:**

```
long ADX_CPW_GetMultipleCharacter(long snum, char *multichar);
```

**CBASIC interface:**

```
SUB Adx.Cpw.GetMultipleCharacter(rc, snum, multichar) EXTERNAL
    INTEGER*4 rc
    INTEGER*4 snum
    STRING multichar
END SUB
```

**Java interface:**

```
(com.ibm.OS4690.security.EnhancedSecurityPasswordSettings)
boolean isMultipleCharacterEnabled();
```

**Return codes:**

- 0 — Success
- ADX\_CPW\_ERR\_SESSION\_TIME\_OUT
- ADX\_CPW\_ERR\_INVALID\_SESSION\_ID
- ADX\_CPW\_ERR\_NOT\_ACTING\_MASTER
- ADX\_CPW\_ERR\_INVALID\_BUFFER

## SetMultipleCharacter

SetMultipleCharacter sets the global setting for the multiple character option to the value indicated by the *multichar* parameter. When the multiple character option is enabled, passwords must contain an upper case character, a lower case character, a numeric character, and a special character. Setting *multichar* to **Y** enables the option, setting it to **N** disables it.

SetMultipleCharacter returns an 80E70022 error if the user is not authorized to modify Enhanced Security Password Settings.

### C interface:

```
long ADX_CPW_SetMultipleCharacter(long snum, char multichar);
```

### CBASIC interface:

```
SUB Adx.Cpw.SetMultipleCharacter (rc, snum, multichar) EXTERNAL
    INTEGER*4 rc
    INTEGER*4 snum
    STRING multichar
END SUB
```

### Java interface:

```
(com.ibm.OS4690.security.EnhancedSecurityPasswordSettings)
void setMultipleCharacterEnabled(boolean multipleCharacterEnabled);
```

### Return codes:

- 0 — Success
- 0x80E7000E — ADX\_CPW\_ERR\_SESSION\_TIME\_OUT
- 0x80E7000F — ADX\_CPW\_ERR\_INVALID\_BUFFER
- 0x80E70011 — ADX\_CPW\_ERR\_INVALID\_SESSION\_ID
- 0x80E70012 — ADX\_CPW\_ERR\_INVALID\_DATA
- 0x80E70018 — ADX\_CPW\_ERR\_NOT\_ACTING\_MASTER
- 0x80E7001C — ADX\_CPW\_ERR\_MIN\_PW\_LENGTH\_MISMATCH
- 0x80E70022 — ADX\_CPW\_ERR\_PASSWORD\_SETTINGS\_AUTHORIZATION\_ERROR

## Commit

This function commits all changes made during the session and then ends the session. If this command returns successfully, all changes made with the given session number (*snum*) are saved permanently.

### C interface:

```
long ADX_CPW_Commit(long snum);
```

### CBASIC interface:

```
SUB Adx.Cpw.Commit(rc, snum) EXTERNAL
    INTEGER*4 rc
    INTEGER*4 snum
END SUB
```

### Java interface:

```
(com.ibm.OS4690.security.EnhancedSecuritySession)
void commit();
```

### Return codes:

- 0 — Success
- ADX\_CPW\_ERR\_SESSION\_TIME\_OUT

ADX\_CPW\_ERR\_INVALID\_SESSION\_ID  
ADX\_CPW\_ERR\_NOT\_ACTING\_MASTER

## Cancel

This function cancels any changes made during the session and then ends the session. All changes made with the given session number (*snum*) are ignored. After a session times out, this call is the only session-related API call that can be made for that session.

### C interface:

```
long ADX_CPW_Cancel(long snum);
```

### CBASIC interface:

```
SUB Adx.Cpw.Cancel(rc, snum) EXTERNAL  
    INTEGER*4 rc  
    INTEGER*4 snum  
END SUB
```

### Java interface:

```
(com.ibm.OS4690.security.EnhancedSecuritySession)  
void cancel();
```

### Return codes:

0 — Success  
ADX\_CPW\_ERR\_INVALID\_SESSION\_ID

---

## Chapter 24. Designing Python Programs for 4690

---

### Python version information

The 4690 version of Python is based on level 2.7.3. Use this version when referencing information from the Python language web site at [www.python.org](http://www.python.org).

---

### Using Python on 4690

- | Python is currently restricted to 4690 enhanced mode store controllers.

### Command mode

The Python interpreter can be launched on a 4690 store controller from command mode via the command `python2` or by the fully qualified name `c:\adx_spgm\python2.386`.

For a brief overview of the command line syntax, run `python2 -?`. For more information on what the switches do and how to run Python programs, see <http://docs.python.org/2/using/cmdline.html>. The command syntax is identical to the syntax of other linux builds of this version of Python, except that additional OS4690 implementation specific flags have been added.

- `-X4690shell` – This causes Python to use the 4690 command processor as the default shell for all core Python functions that launch shell programs. This flag will cause issues if used with third party Python code that runs commands via a command shell and is unaware of the flag's effect. This switch affects the following functions:
  - `os.system()`
  - The `popen` class when the ***cmd*** parameter is a string
  - The `subprocess` class when the `shell` parameter is `True`
- | • `-Xswitch` – Switches to the enhanced mode graphical screen when launching a Python program. The screen switch will only happen when Python is running as a foreground application. The `-Xswitch` command must come before all non-extended switches (i.e. those that begin with `-X`)
- | • `-X@` - This allows environment variables to be specified on the command line. The embedded linux environment that runs the Python interpreter allows for much longer names in environment variables and is case specific. This option allows linux environment variables like `PYTHONPATH` to be used more easily.

**Note:** Capital NAME and VALUE are used for emphasis – actual environment variables names are case sensitive therefore `NAME ≠ Name ≠ name`.

- `-X@NAME` removes the environment variable `NAME`
- `-X@NAME=` sets the environment variable `NAME` to an empty string:
- `-X@NAME=VALUE` sets the environment variable `NAME` to `VALUE`

Launching Python without a command, module name, or file name will start the interpreter in interactive mode. Using the `-i` flag will enable interactive mode even in these cases. If `stdin` is a console, the `gnu readline` library will be used to read user input. When Python is run from a 4690 command window, `readline` is configured so that the key combinations are more familiar to 4690 users. The configured keys are listed below. When Python is launched using any other method, the default `readline` keyboard configuration (`emacs`) is used. For more information see the documentation link at [www.gnu.org/software/readline/](http://www.gnu.org/software/readline/).

### Background Application

Python programs may be configured as background applications using the normal 4690 OS controller configuration procedure. Care should be taken when using this option because standard input and standard error streams will be discarded when running in background mode. This means that program

output and error information will be lost unless the Python program being started takes some action. You may also avoid this by running python2 from a BAT file and starting the BAT file as a background task. Using a BAT file is also required if the length of the command line exceeds the limit imposed in configuration.

To configure a Python background application:

| Program Name: `adx_spgm:python2.386`

Parameter List: (see the command line parameters in “Command mode” on page 713).

## **Other Launch Mechanisms**

| Python programs may also be launched from batch files and other compiled programs on 4690. Launching from Java programs is also possible but remember to use 4690Runtime classes in Java-6 to launch  
| `adx_spgm:python2.386`.

## **4690 OS Terminal Support Statement**

| Running Python programs on a 4690 terminal is not supported.



---

## Standard Library Functions

The following categories of functions reference the Python Standard Library document sections for Python 2.7.3.

Table 64. Python Standard Library Functions

Built-in functions	All Supported <b>Note:</b> Import by filename is not supported.	abss() divmod() input() open() staticmethod() all() enumerate() int() ord() str() any() eval() isinstance() pow() sum() basestring() execfile() issubclass() print() super() bin() file() iter() property() tuple() bool() filter() len() range() type() bytearray() float() list() raw_input() unichr() callable() format() locals() reduce() unicode() chr() frozenset() long() reload() vars() classmethod() getattr() map() repr() xrange() cmp() globals() max() reversed()
--------------------	--	---

Table 64. Python Standard Library Functions (continued)

		zip() compile() hasattr() memoryview() round() __import__() complex() hash() min() set() apply() setattr() help() next() setattr() buffer() dict() hex() object() slice() coerce() dir() id() oct() sorted() intern()
Built-in types	All Supported	
Built-in exceptions	All Supported	
Built-in constants	All Supported	
String Services	All Supported	string re struct difflib StringIO cStringIO textwrap codecs unicodedata stringprep fpformat

Table 64. Python Standard Library Functions (continued)

Data types	All Supported	datetime calendar collections heapq bisect array sets sched mutex Queue weakref UserDict UserList UserString types new copy pprint repr
Numeric and Mathematical Modules	All Supported	numbers math cmath decimal fractions random itertools functools operator
File and Directory Access	Some Supported (see also this document section explaining special considerations for files and directories on 4690 OS) Note that <i>glob</i> and <i>fnmatch</i> always work with linux style file names and naming rules.	os.path fileinput stat statvfs filecmp tempfile glob fnmatch linecache shutil dircache
Data Persistence	DBM related packages are not supported on 4690 files systems. Use linux style path names like <b>/home/vxuser</b> with these modules.	pickle cPickle copy_reg shelve marshal anydbm whichdb dbm gdbm dbhash bsddb dumpdbm sqlite
Data Compression and Archiving	All Supported functionally, however archives containing links and long file names cannot be decompressed to 4690 file systems that do not support them.	zlib gzip bz2 zipfile tarfile

Table 64. Python Standard Library Functions (continued)

File Formats	All Supported	csv ConfigParser robotparser netrc xdrlib
Cryptographic Services	All Supported	hashlib hmac md5 sha
Generic OS Services	All Supported	os io time argparse optparse getopt logging logging.config logging.handlers getpass curses curses.textpad curses.ascii platform errno ctypes
Optional OS Services	Some Supported	select threading thread mmap (restricted to F:) readline rlcompleter
Interprocess Communication and Networking	All Supported	subprocess socket ssl signal popen2 asyncore asynchat
Internet Data Handling	Mail related packages are not supported on 4690 file systems. Other packages are supported.	email json mailcap mailbox mhtml mimetools mimetypes MimeWriter mimify multifile rfc822 base64 binhex binascii quopri uu

Table 64. Python Standard Library Functions (continued)

Structured Markup Processing Tools	All Supported	HTMLParser sgmllib htmllib htmlentitydefs xml.etree.ElementTree xml.dom xml.com.minidom xml.dom.pulldom xml.sax xml.sax.handler xml.sax.saxutils xml.sax.xmlreader
Internet Protocols and Support	All Supported	webbrowser cgi cgiitb wsgiref urllib urllib2 httplib ftplib poplib imaplib nntplib smtplib smtpd telnetlib uuid urlparse SocketServer BaseHTTPServer SimpleHTTPServer CGIHTTPServer cookielib Cookie xmlrpclib SimpleXMLRPCServer DocXMLRPCServer
Multimedia Services	Audio support for Python is currently not present in this release	audioloop imageop aifc sunau wave chunk coloursys imghdr sndhdr ossaudiodev
Internationalization	All Supported	gettext (see notes) locale (see notes)
Program Frameworks	All Supported	cmd shlex

Table 64. Python Standard Library Functions (continued)

Graphical User Interfaces	Some Supported. <b>Note:</b> pyGTK is currently not enabled in this release.	Tkinter tkk Tix ScrolledText turtle IDLE pyGTK – Not supported
Development Tools	All Supported but <b>doctest</b> may require long file name file systems	pydoc doctest unittest test test.test_support
Debugging and Profiling	Not supported	bdb pdb hotshot timeit trace
Python Runtime Services	All Supported, but <b>distutils</b> has some limitations based on the lack of other supporting packages like RPM and GCC	sys sysconfig __builtin__ future_builtins __main__ warnings contextlib abc atexit traceback __future__ gc inspect site user fpectl distutils
Custom Python Interpreters	All Supported	code codeop
Restricted Execution	All Supported	rexec Bastion
Importing Modules	All Supported	imp importlib imputil zipimport pkgutil modulefinder runpy

Table 64. Python Standard Library Functions (continued)

Python Language Services	All Supported	parser ast symtable symbol token keyword tokenize tabnanny pyclbr py_compile compileall dis pickletools
Miscellaneous Services	All Supported	formatter
MS Windows Specific Services	None Supported	
Unix Specific Services	All Supported	posix pwd spwd grp crypt dl termios tty pty fcntl pipes posixfile resource syslog commands
Mac OS X Specific Services	None Supported	
MacPython OSA Modules	None Supported	
SGI IRIX Specific Services	None Supported	
SunOS Specific Services	None Supported	

## Working with Files and External Programs on 4690

The 4690 environment includes multiple file systems with different and conflicting rules. For example, the C: drive is restricted to 8.3 style file names and is case-insensitive, the F: drive supports long file names and is case-sensitive, and the M: drive supports long file names and stores files with case intact but operates with certain case-insensitive rules. In addition to this, the 4690 Python implementation is aware of 4690 file systems and the underlying embedded Linux file system. As a result, more care may be required when working with files and external programs on 4690 than on other platforms that have predominate rules for all file systems.

## Running programs on 4690

The python2.386 program is a 4690 program, but python2 is a linux program. The process 4690 knows as python2 acts as a proxy for the linux process (interfacing with 4690 and doing I/O on its behalf). This has implications for several of process control functions relative to running linux programs directly:

- The `os.exec` functions will not cause the python2 proxy to end (the proxy will remain running while the new linux process runs underneath it).

- Each linux process run via a proxy starts in a unique process group. Any processes still in this process group will be terminated when the proxy ends.

The current drive and directory of 4690 is not changed when Python's current directory is changed. The current directory of all 4690 commands run by Python will be initialized to the current directory when python2.386 was launched.

```
| There are several methods available for launching 4690 programs. If python2 was started with the
| -X4690shell command line parameter then os.system( ) will launch all programs via the 4690 command
| shell. Likewise specifying shell=True on Popen will also assume the 4690 command shell should be used.
| Since this may break other packages, the os4690.system( ) function can be used anytime to launch a
| command via the 4690 shell. It is also possible to use os4690.resolvepath( ) to retrieve the linux
| equivalent file name for an executable program and use that directly in Popen and other subprocess
| modules. An example of this latest method is available in the 4690 OS supplemental package Python
| sample programs (see cfg_util/Runner.py launching the 4690 Java-2 jvm to run the 4690 Java XML
| configuration utility). See file 4690opt/readop.doc on the 4690 OS CD for more information about installing
| the optional program packages. This sample code is intended to be explanatory and will not be useful
| without tailoring.
```

Some programs like Java-6 and Python itself can be run from either the embedded linux or 4690 environments. Often there is a trade-off between convenience and performance that depends on how the program is launched. For example, launching Java-6 programs directly as /usr/bin/java requires you to set up the class path and other environment variables, whereas launching through 4690 as javaebn:java will setup the class path as per configuration. Process to process communication is much faster when launched directly because communication streams do not need to route through the 4690 environment.

## Running embedded linux programs

Launching linux side programs works as documented in the Python guides but there are a few things to keep in mind:

- Python programs run as an unprivileged (non-root) user vxuser. The home directory for this user is /home/vxuser which is also accessible as f:/adxetc/home/vxuser when booted normally. The home directory /home/vxuser is hosted on a temporary ram drive that is not accessible from 4690 side applications when booted from supplementals.
- Other linux programs like bash, sed, awk, etc will not recognize 4690 file names passed to them as parameters.
- 4690 does not provide a general purpose linux distribution. There is no guarantee that a specific linux side tool will be available or will continue to be available in future releases of 4690. As a best practice, limit use of linux side tools to those that are most common and to tools that you provide and therefore have control over.

### Examples:

This will fail because the external tar program cannot use 4690 style file names:

```
import os
os.system('tar -czf c:/test.tgz -C c:/ adx_upgm')
```

#### Alternative 1: Use built in methods

```
import tarfile
with tarfile.open('c:/test.tgz','w:gz') as tf:
    tf.add(name='c:/adx_upgm',arcname='adx_upgm')
```

#### Alternative 2: Resolve path names before calling external program

```
from os4690 import resolvepath as fn
from subprocess import call
call(['tar', '-czf', fn('c:/test.tgz'), '-C', fn('c:/'), 'adx_upgm'])
```



## Rules for Path Names in the 4690 implementation of Python:

- The 4690 implementation of Python treats both slash “/” and back-slash “\” as path separators, regardless of whether the file or command being referenced is contained in a 4690 file system or in a Linux file system.
- The characters slash “/”, back-slash “\”, colon “:”, and wild card characters “\*” and “?” are all reserved under 4690. It is possible for file names that contain these characters to be created by other programs on some 4690 file systems, such as on F: (which is a linux filesystem). However, attempting to access paths, commands, or files that contain these restricted characters from 4690 may produce unpredictable results therefore try to avoid such naming conventions.
- The Python implementation classifies paths as one of the three following types and processes each using different rules.
  - 4690 Paths – Any path that has a colon before the first slash or backslash or a colon with no slash of either kind is considered a 4690 path. These paths are first expanded resolved using standard 4690 name resolution methods. Logical names are expanded and the current directory and drive are used to generate a fully qualified 4690 path. This path is then mapped to the linux filesystem. For example, the path C:/ will be mapped to /opt/ibm/retail/vx4690/device/fuse/c\_drive. If a 4690 path is not accessible from linux (for example x::y::z), then a known invalid path name is returned.
  - Absolute Paths – Path names starting with slash “/” or back-slash “\” (which is equivalent as noted above) are absolute Linux style path names. They are passed directly to the underlying OS.
  - Relative paths – Any other path is a relative path. This includes paths containing colons (as long as a slash or backslash precedes the colon in the path name). Relative paths are converted to absolute paths by prepending the current working directory name to the given path and then normalizing it (as in `os.path.normpath`). This means that 4690 logical file names like *hosts* will not automatically be recognized as such.
- `Callingisabs()` on a 4690 path will always return true.
- Relative path names are always relative to the current working directory.
- File list functions like `os.getcwd()` always return a Linux style name. This means that `os.chdir("C:/")` followed by `result=os.getcwd()` will produce `result="/opt/ibm/retail/vx4690/device/fuse/c_drive"` and not `result="C:/"`.
- The `fnmatch` function and by extension the `glob` module functions normally match case on Linux systems and ignore case on Microsoft Windows systems. Neither behavior is completely appropriate for 4690 since the file systems and not the OS determine case sensitivity. However, one or the other default behavior must be chosen, so for that purpose the Python implementation on 4690 will use case-sensitive matching for `fnmatch` (and `glob`). The effect is mitigated somewhat because case-insensitive file systems like C: and D: will always return file and directory names in lower case.

## 4690 Python API

The 4690 version of Python has a built in module called `os4690` which serves as the basis for additional `os4690` functionality in Python. The module's `__all__` list indicates the publicly usable parts of the module. The following additional functions are provided:

`getln(lname,table=LN_BOTH,resolve_fully=False) -> string`

Returns the value of a 4690 logical name string (`lname`). The `table` variable indicates which logical table to search. The possible values are:

- `LN_PROCESS` - Search the current process' table
- `LN_SYSTEM` - Search the system table
- `LN_BOTH` - Search the process table then the system table

If `resolve_fully` is `False` a single lookup only is done. Otherwise the result of each lookup is fetched again (from the same tables as the first) until the name is no longer found. When used with `LN_BOTH`, this emulates how 4690 resolves logical names when doing filename resolution.

Returns value of the logical name or `None` if the logical name is not set.

**isavailable()** -> boolean

Returns True if the 4690 fuse filesystem and other services are available and False otherwise. Unless there is a problem or 4690 is not running this will always return True.

When 4690 API calls are not available, most functions in this module will not work. In addition, the standard file and path access methods in Python will not properly resolve 4690 path names nor run 4690 programs.

**resolvepath(path)** -> linuxpath

Resolves a given 4690 path to a fully qualified linux path. The name is resolved relative to the current drive and directory where Python was started. Logical names without colons are also resolved using this method. For example, `resolvepath("hosts")` will return a linux path that refers to the 4690 hosts file in the ADX\_SDT1: directory. If the given 4690 path is invalid or is NOT accessible from linux, this function will return None.

**system(command)** -> exit\_status

Executes the given command in a 4690 command shell. This is useful when you wish to run occasional commands in 4690 but don't want the possible problems of using the `-X4690shell` flag. Alternatively, you may use the constant `os4690.SHELL_PATH` to refer to the linux path to the 4690 command interpreter. Only the least significant 7 bits of the program exit code will be returned (0 – 127).

**tz()** -> timezone

Returns a linux style time zone based on the time zone as set in 4690. Note that Python does not provide a default timezone implementation. The time zone identifier returned from this function may be useful with additional package content such as that provided by “pytz”.

---

## Differences Between Console and Graphical Output

The 4690 text mode console receives standard output and standard error streams from Python programs that were not launched from background application control. Writing text to the standard output stream is similar to writing “Echo” statements in a batch file.

It is possible to create graphical user interfaces with the level of Python initially delivered, but no window control API's are available at this time. Python graphical windows are created on the enhanced mode graphical extensions screen and share this space with other applications like the 4690 web browser. This screen is accessible on the System Keys menu via `Alt+SysRq – X`. Unlike with Java-6, when graphical windows are created there is no automatic switch to the new screen. You may use the switch `-Xswitch` to cause Python to switch to the enhanced mode graphical application screen when it is started. The window will switch even if Python is started in interactive mode.

It should be noted, however, that a number of additional packages like PyGTK are not provided with this version of 4690 OS and missing dependencies may limit the tool sets that can be used to create GUI programs. Additional base packages may be added in future releases.

---

## Installing Third Party Packages

The Python Package Index at [pypi.python.org](http://pypi.python.org) lists a number of additional packages that may be useful in creating more complex Python applications. Pure Python packages without dependencies are the most likely to be useful with 4690 since the OS provides only a limited subset of support packages beyond the base Python libraries at the current level of 4690 OS.

Installing on a local test system can be accomplished by running the `setup.py` script provided with the source distribution, but because `python.386` runs as unprivileged, user write access to the global `site-packages` is not available. Instead, pass the “--user” option on the command line to install the package

into the vxuser home Python site-packages directory by entering: `python2 setup.py install --user`. This user level site-packages directory is searched by default when importing modules.

Installing packages remotely can be accomplished with the controller preload utility. The format of the preload command file is `ADXC?T?Z.DAT` where the 5th and 7th letter follow the same conventions as for 4690 ASM product control files. The following example uses `ADXCPTAZ.DAT` as the name for the preload command file and supposes the `pytz` (a Time zone implementation) will be the package to be installed. If `ADXCPTAZ.DAT` is already in use by another product in your environment, use a different name.

#### | **Extra packages zip file container ADX\_UPGM:ADXCPTAZ.ZIP**

Create a zip file containing (for this example) `pytz-2013b.tar.gz`

```
Archive:  adxcptaz.zip
  Length      Date    Time    Name
-----
 331277    05-17-2013  11:28    pytz-2013b.tar.gz
-----
 331277                                1 file
```

#### | **Preload Command File ADX\_UPGM:ADXCPTAZ.DAT**

```
# Install extra Python packages
trigger changed=adx_upgm:adxcptaz.zip mode=e
msg -1 Installing extra Python packages
run adx_spgm:command.286 -c adx_upgm:adxcptaz.bat
```

#### | **Preload Batch File ADX\_UPGM:ADXCPTAZ.BAT**

```
REM The only purpose for the batch file is to run the
REM Python script with stderr redirected to
REM a file to catch errors.
python2 adx_upgm:adxcptaz.py >* adx_upgm:adxcptaz.log > adx_upgm:adxcptaz.log
```

#### | **Extract and Install Script ADX\_UPGM:ADXCPTAZ.PY**

```
# Python extra packages example installer script
import zipfile
import tarfile
import shutil
import sys
import os

# Working directory
TEMP_DIR='/tmp/adxcptaz'

# Zip file with extra Python packages
ZIP_FILE='c:/adx_upgm/adxcptaz.zip'

# Delete working directory if it exists
if os.path.exists(TEMP_DIR):
    shutil.rmtree(TEMP_DIR)

# Create working directory and change to it
os.mkdir(TEMP_DIR)
os.chdir(TEMP_DIR)

# Extract the extra packages from the zip file archive
with zipfile.ZipFile(ZIP_FILE,'r') as z:
    print '+++++'
    print 'Unzipping ' + ZIP_FILE
    sys.stdout.flush()
    z.extractall()

# Find the tgz and tar.gz files we just extracted
for fileName in os.listdir('.'):
    pass
```

```

if os.path.isfile(fileName):
    if fileName.endswith('tgz') or fileName.endswith('gz'):
        with tarfile.open(fileName, 'r') as t:
            print '++++++'
            print 'Untar ' + fileName
            sys.stdout.flush()
            t.extractall()

# The tar files created subdirectories with setup.py scripts
# we need to run to complete the installation
for dirName in os.listdir('.'):
    if os.path.isdir(dirName):
        print '++++++'
        print 'Installing ' + dirName
        sys.stdout.flush()
        # Change to the subdir and run the setup.py script provided
        # by the package with the --user switch to put the files in
        # the writable per-user site packages directory
        os.chdir(dirName)
        os.system('python2 setup.py install --user')
        os.chdir(TEMP_DIR)

# Cleanup
os.chdir('/')
shutil.rmtree(TEMP_DIR)

# Done
print '++++++'
print ' '
print 'Done.'
sys.stdout.flush()
sys.exit(0)

```

---

## Limitations

See individual limitations in the package listing above.

- | Python access to 4690 file systems is not available during any IPL Command Processor stage.
- | Opening files across the network, such as by the use of node ID's like 'ADXLXAAN::' is not supported.
- | Methods to perform keyed file reads and write or access to ADXSERVE programming interfaces are not included.

---

## Appendix A. Operating system disk directory

The operating system supports a hierarchical directory structure. This structure begins with the root directory on each hard disk or diskette. The root directory may contain entries for files or entries for more directories. Each directory may contain entries for files or more directories. The directories beyond the root are referred to as subdirectories. This hierarchical structure enables a file to be placed in the root or in any subdirectory. The sequence of subdirectories starting with the root and proceeding to the subdirectory containing the file is called the *subdirectory path*.

The operating system has a predefined set of subdirectories for the use of the operating system and the application programs. You may also define additional subdirectories for other uses. All subdirectories predefined for the operating system are defined from the root directory. This means that they are only one level deep. These subdirectories are created on the C drive during the installation process.

To use the 4690 Apply Software Maintenance procedures, you must have a set of three subdirectories on the same drive. The names for these subdirectories must have the same first five characters and must end with PGM, MNT, and BUL. The names must begin with ADX\_, and they must be defined as logical subdirectory names in the logical names files.

---

### Naming conventions for operating system files

The files that are added to the operating system are named using this convention:

ADXxxxx.eee or VX\_xxxx.eee

where:

**ADX** = Operating system prefix  
**xxxxx** = Remainder of file name (5 characters)  
**eee** = File extension (0 to 3 characters)  
**VX\_** = V6 Enhanced Mode file version

Base files from the operating system do not follow this format (for example, CHKDSK.286).

Most ADX files follow another convention in which the last character of the file name and the file extension identify the type of file. For more information, see “Rules for naming subdirectories and files on FAT systems” on page 17.

---

### Dictionary of operating system files

Table 65 lists the names of predefined subdirectories, the directory in which each is located, and a description of the contents of the subdirectory.

*Table 65. Dictionary of predefined subdirectories and their contents*

Subdir.	Directory	Contents
ADX_BSX	root	Symbol files.
ADX_CLOG	root	Log files for command line and ADXCSJ0L activity.
ADXETC\JAVA2\CORE	m:\	Java core files for Java 2 programs as documented in Java 2.
ADXETC\JAVA2\DUMP	m:\	Files used to implement the Java 2 JVM dump facility (not currently used on 4690).
ADXETC\JAVA2\PREFS	m:\	Files used to store preferences information for the Java 1.4 preferences API.
ADXETC\JAVA2\TMP	m:\	Default location for the java.io.tmpdir property in Java 2.

Table 65. Dictionary of predefined subdirectories and their contents (continued)

Subdir.	Directory	Contents
ADXETC\JAVA2\TOF	m:\	Files used for the Terminal Offline Function (TOF) with Java 2.
ADX_IBUL	root	Backup level of maintenance modules for application programs. Modules that were replaced from ADX_IPGM by the modules in ADX_IMNT by Apply Software Maintenance.
ADX_IDT1	root	Data files used by the application programs. This subdirectory is used on every drive.
ADX_IDT4	root	Data files used by the application programs. This subdirectory is used on every drive.
ADX_IMNT	root	Maintenance modules for application programs. Maintenance modules for the modules in ADX_IPGM that are to be maintained by Apply Software Maintenance.
ADX_IOSS	root	Print spooler data and controls.
ADX_IPGM	root	Active application programs. Terminal and store controller application programs and associated files for messages, display screens, configuration data, and so on. This subdirectory is used for the programs that provide store checkout and accounting functions. If the Toshiba Licensed Program for the 4690 Store System is used, it will be placed in the set of ADX_lxxx subdirectories.
ADXJ2CCL	root	Copies of various files from the current Java 2 installation. Used for building the loadshrink image for the terminal.
ADX_KBUL	root	Backup level for ADX_KMNT.
ADX_KDT1	root	Data files for SSRT.
ADX_KMNT	root	Maintenance modules for ADX_KPGM.
ADX_KPGM	root	SSRT code and code related data.
ADX_SBUL	root	Backup level of maintenance modules for system programs. Modules that were replaced from ADX_SPGM by the modules in ADX_SMNT by Apply Software Maintenance.
ADX_SBUN	root	Backup level of NLS translation file feature.
ADX_SDT1	root	Data files used by the system programs. This subdirectory is used on every drive.
ADX_SMNT	root	1) Maintenance modules for system programs. Maintenance modules for the modules in ADX_SPGM that are to be maintained by Apply Software Maintenance. 2) Active terminal preload data files.
ADX_SMNT/INACTIVE	root	Inactive terminal preload data files. Files are placed in this directory during the rebuild process, then moved to the active preload directory after a successful rebuild.
ADX_SPGM	root	Active system programs. Terminal and store controller system programs and associated files for messages, display screens, configuration data, and so on.
ADX_STLD	root	Active terminal preload files. These files are used by the Terminal Preload Rebuild Utility.

Table 65. Dictionary of predefined subdirectories and their contents (continued)

Subdir.	Directory	Contents
ADX_STLD\INACTIVE	root	Inactive terminal preload files. These files are used by the Terminal Preload Rebuild Utility.
ADX_UBUL	root	Backup level of maintenance modules for user programs. Modules that were replaced from ADX_UPGM by the modules in ADX_UMNT by Apply Software Maintenance.
ADX_UDT1	root	Data files used by the user programs.
ADX_UMNT	root	Maintenance modules for your applications. This subdirectory is used on every drive. Maintenance modules for the modules in ADX_UPGM that are to be maintained by Apply Software Maintenance.
ADX_UPGM	root	Active user programs. This subdirectory should be used for program development activities in a development environment or may be used for additional programs in a store environment. The HCP names supported for files in this subdirectory have a different structure than the names for the ADX_Sxxx and ADX_lxxx subdirectories.
JAVA	root	4690 Class files.
JAVA\LIB	root	Java class and profile files.
JAVA2	m:\	Files used by Java 2.
JAVA2\BIN	m:\	Java 2 Software Development Kit (SDK) programs and DLLs.
JAVA2\DOCS	m:\	Any documentation files shipped with Java 2.
JAVA2\INCLUDE	m:\	Header files used with JNI in Java 2.
JAVA2\JRE\BIN	m:\	Programs and DLLs that are part of the Java 2 Java Runtime Environment (JRE).
JAVA2\JRE\BIN\ CLASSIC	m:\	The Java 2 JVM.
JAVA2\JRE\LIB	m:\	JAR files and property files used by the Java 2 JRE.
JAVA2\JRE\LIB\AUDIO	m:\	Audio files that are part of the Java 2 implementation. <b>Note:</b> Playing audio is not supported on 4690 OS.
JAVA2\JRE\LIB\CMM	m:\	Files used by the color management classes in Java 2.
JAVA2\JRE\LIB\EXT	m:\	Java 2 extension JAR files.
JAVA2\JRE\LIB\FONTS	m:\	Java 2 font files (true-type format).
JAVA2\JRE\LIB\IMAGES\CURSORS	m:\	Drag and drop cursors for Java 2.
JAVA2\JRE\LIB\ SECURITY	m:\	Java 2 security package related files.
JAVA2\LIB	m:\	JAR and library files used with the Java 2 SDK.

The current directory of 4690 OS file names is available on the Toshiba support site (use the Knowledgebase search field).



---

## Appendix B. Error Messages

This appendix contains error messages that occur while using the library utility (LIB86), the POSTLINK utility (POSTLINK), and the linker utility (Link86).

---

### LIB86 Error Messages

LIB86 can produce error messages during processing. With each message, LIB86 displays additional information appropriate to the error, such as the file name or module name, to help isolate the location of the problem. This section describes each error message, its cause, the library action, and the user response.

---

#### CANNOT CLOSE:

**Explanation:** LIB86 cannot close an output file.

**System action:** LIB86 terminates and displays this message, followed by the name of the file.

**User response:** Make sure the correct disk is in the drive and that it is not write-protected.

---

#### DIRECTORY FULL:

**Explanation:** There is not enough directory space for the output files.

**System action:** LIB86 terminates and displays this message.

**User response:** Erase unnecessary files or use a disk with fewer files.

---

#### DISK FULL:

**Explanation:** There is not enough disk space for the output files.

**System action:** LIB86 terminates and displays this message.

**User response:** Erase unnecessary files or use a disk with more space.

---

#### DISK READ ERROR:

**Explanation:** LIB86 detects a disk error while reading the indicated file.

**System action:** LIB86 terminates and displays this message, followed by the name of the file.

**User response:** Try to regenerate the file.

---

#### INVALID COMMAND OPTION:

**Explanation:** LIB86 encounters an unrecognized option in the command line.

**System action:** LIB86 terminates and displays this message.

**User response:** Retype the command line or edit the INP file.

---

#### LIB86 ERROR 1:

**Explanation:** Internal LIB86 error.

**System action:** LIB86 terminates and displays this message.

**User response:** Copy the library, the files you are trying to add, replace, and so on, (if any) onto a diskette. Then run the library command again, but add the following text to the end of the command line, *preceded by a single blank space:*

>filename.ext

This creates a file that captures the console output of the library utility. Copy filename.ext onto the same diskette with the library and other files. Contact your service representative when you have gathered the above information.

---

#### MODULE NOT FOUND:

**Explanation:** The indicated module name, which appeared in a REPLACE, SELECT, or DELETE command line option, could not be found.

**System action:** LIB86 terminates and displays this message, followed by the name of the module.

**User response:** Retype the command line, or edit the INP file.

---

#### MULTIPLE DEFINITION:

**Explanation:** The indicated symbol is defined as PUBLIC in more than one module.

**System action:** LIB86 displays this message, followed by the name of the symbol.

**User response:** Correct the problem in the source file, regenerate the object file, and try again.

## Error Messages

---

### NO FILE:

**Explanation:** LIB86 could not find the indicated file.

**System action:** LIB86 terminates and displays this message, followed by the name of the file.

**User response:** Correct the file name or drive identifier and try again.

---

### OBJECT FILE ERROR:

**Explanation:** LIB86 detected an error in the object file. This is caused by a compiler error, or a bad disk file.

**System action:** LIB86 terminates and displays this message, followed by the name of the file.

**User response:** Regenerate the object file and try again. If the error still occurs, copy the source file and the object file onto a diskette. Then run the library command again, but add the following text to the end of the command line, ***preceded by a single blank space:***

```
>filename.ext
```

This creates a file that captures the console output of the library utility. Copy filename.ext onto the same diskette with the library and other files. Contact your service representative when you have gathered the above information.

---

### RENAME ERROR:

**Explanation:** LIB86 cannot rename a file.

**System action:** LIB86 terminates and displays this message.

**User response:** Make sure the disk is not write-protected, and then try again.

---

### SYMBOL TABLE OVERFLOW:

**Explanation:** There is not enough memory for the symbol table.

**System action:** LIB86 terminates and displays this message.

**User response:** Reduce the number of command line options in the command line (MAP and XREF both use symbol table space), or use a system with more memory.

---

### SYNTAX ERROR:

**Explanation:** LIB86 detected a syntax error in the command line, probably due to an improper file name or a command option that is not valid.

**System action:** LIB86 stops, displays this message, and echoes the command line up to the point where it found the error.

**User response:** Retype the command line or edit the INP file, and try again.

---

## POSTLINK Error Messages

During the course of operation, POSTLINK can display error messages. This section describes each error message, its cause, the action taken by the postprocessor utility, and the user response.

---

### CANNOT ERASE OLD FILE:

**Explanation:** POSTLINK was started and the input file was open to another process.

**System action:** POSTLINK terminates and displays error message.

**User response:** Correct the error by rerunning the linker and then the postlinker.

---

### CANNOT READ FIXUPS:

**Explanation:** POSTLINK was started and a disk read error occurred.

**System action:** POSTLINK terminates and displays error message.

**User response:** Correct the error by rerunning the linker and then the postlinker.

---

### CANNOT RENAME FILE:

**Explanation:** POSTLINK was started and the input file was open to another process.

**System action:** POSTLINK terminates and displays error message.

**User response:** Correct the error by rerunning the linker and then the postlinker.

---

### CANNOT SEEK TO FIXUPS:

**Explanation:** POSTLINK was started and a disk seek error occurred.

**System action:** POSTLINK terminates and displays error message.

**User response:** Correct the error by rerunning the linker and then the postlinker.

---

### COULD NOT OPEN FILE filename.ext

**Explanation:** POSTLINK was started and could not open the given file. This is because the file does not exist, or the user does not have authorization to use POSTLINK.286 or create files in the subdirectory where POSTLINK or the specified file resides.

**System action:** POSTLINK terminates and displays error message.

**User response:** Correct the error by keying in POSTLINK and the file name.

#### **COULD NOT OPEN POSTOUT.286:**

**Explanation:** POSTLINK was started and a disk open error occurred.

**System action:** POSTLINK terminates and displays error message.

**User response:** Correct the error by rerunning the linker and then the postlinker.

#### **COULD NOT READ FILE:**

**Explanation:** POSTLINK was started and a disk read error occurred.

**System action:** POSTLINK terminates and displays error message.

**User response:** Correct the error by rerunning the linker and then the postlinker.

#### **COULD NOT READ HEADER:**

**Explanation:** POSTLINK was started and a disk read error occurred.

**System action:** POSTLINK terminates and displays error message.

**User response:** Correct the error by rerunning the linker and then the postlinker.

#### **COULD NOT READ OLD BUCKET:**

**Explanation:** POSTLINK was started and a disk read error occurred.

**System action:** POSTLINK terminates and displays error message.

**User response:** Correct the error by rerunning the linker and then the postlinker.

#### **COULD NOT READ SRTL HDR:**

**Explanation:** POSTLINK was started and a disk read error occurred.

**System action:** POSTLINK terminates and displays error message.

**User response:** Correct the error by rerunning the linker and then the postlinker.

#### **COULD NOT SEEK TO BEGINNING OF FILE:**

**Explanation:** POSTLINK was started and a disk seek error occurred.

**System action:** POSTLINK terminates and displays error message.

**User response:** Correct the error by rerunning the linker and then the postlinker.

#### **COULD NOT SEEK TO READ OLD BUCKET:**

**Explanation:** POSTLINK was started and a disk seek error occurred.

**System action:** POSTLINK terminates and displays error message.

**User response:** Correct the error by rerunning the linker and then the postlinker.

#### **COULD NOT SEEK TO SRTL HDR:**

**Explanation:** POSTLINK was started and a disk seek error occurred.

**System action:** POSTLINK terminates and displays error message.

**User response:** Correct the error by rerunning the linker and then the postlinker.

#### **COULD NOT WRITE BUCKET:**

**Explanation:** POSTLINK was started and a disk write error occurred.

**System action:** POSTLINK terminates and displays error message.

**User response:** Correct the error by rerunning the linker and then the postlinker.

#### **COULD NOT WRITE HEADER TO OUTPUT FILE:**

**Explanation:** POSTLINK was started and a disk write error occurred.

**System action:** POSTLINK terminates and displays error message.

**User response:** Correct the error by rerunning the linker and then the postlinker.

#### **COULD NOT WRITE LDT:**

**Explanation:** POSTLINK was started and a disk write error occurred.

**System action:** POSTLINK terminates and displays error message.

**User response:** Correct the error by rerunning the linker and then the postlinker.

## Error Messages

---

### COULD NOT WRITE SRTL:

**Explanation:** POSTLINK was started and a disk write error occurred.

**System action:** POSTLINK terminates and displays error message.

**User response:** Correct the error by rerunning the linker and then the postlinker.

---

### COULD NOT WRITE TO FILE:

**Explanation:** POSTLINK was started and a disk write error occurred.

**System action:** POSTLINK terminates and displays error message.

**User response:** Correct the error by rerunning the linker and then the postlinker.

---

### FILE ALREADY POST PROCESSED:

**Explanation:** POSTLINK was started and found the POSTLINK step has run for this file.

**System action:** POSTLINK terminates and displays error message.

**User response:** Correct the error by rerunning the linker and then the postlinker.

---

### FIXUP # TOO LARGE:

**Explanation:** POSTLINK was started and an incorrect fixup record was found.

**System action:** POSTLINK terminates and displays error message.

**User response:** Correct the error by rerunning the correct linker for your compiler.

---

### INVALID FIXUP:

**Explanation:** POSTLINK was started and an incorrect fixup record was found.

**System action:** POSTLINK terminates and displays error message.

---

## LINK86 Error Messages

During the course of operation, LINK86 can display error messages. This section describes each error message, its cause, the linker action, and the user response.

---

### ALIGN TYPE NOT IMPLEMENTED:

**Explanation:** The object file contains a segment align type not implemented in LINK86. The object file is incompatible with LINK86, possibly because it was produced by an unsupported compiler or assembler, or because it was corrupted.

**User response:** Correct the error by rerunning the correct linker for your compiler.

---

### INVALID INPUT FILE:

**Explanation:** POSTLINK was started and the header of the input file was not valid.

**System action:** POSTLINK terminates and displays error message.

**User response:** Correct the error by rerunning the correct linker for your compiler.

---

### SEEK ERROR READING FILE:

**Explanation:** POSTLINK was started and a disk seek error occurred.

**System action:** POSTLINK terminates and displays error message.

**User response:** Correct the error by rerunning the linker and then the postlinker.

---

### SEEK ERROR WRITING TO FILE:

**Explanation:** POSTLINK was started and a disk seek error occurred.

**System action:** POSTLINK terminates and displays error message.

**User response:** Correct the error by rerunning the linker and then the postlinker.

---

### WRONG NUMBER OF PARAMETERS:

**Explanation:** POSTLINK was started and more than one argument was given.

**System action:** POSTLINK terminates and displays error message.

**User response:** Correct the error by keying in POSTLINK and the file name.

**System action:** LINK86 terminates and displays this error message.

**User response:** Correct the object file and restart LINK86 or use the correct linker for your compiler.

---

### CANNOT CLOSE:

**Explanation:** LINK86 cannot close an output file.

**System action:** LINK86 terminates and displays this error message, followed by the name of the file.

**User response:** Restart LINK86 after making sure the correct disk is in the drive and that it is not write-protected.

---

#### CLASS NOT FOUND:

**Explanation:** The class name specified in the command line does not appear in any of the files linked.

**System action:** LINK86 terminates and displays this error message, followed by the name of the missing class.

**User response:** Restart LINK86 with the correct class name in the command line or change the class name in the files being linked.

---

#### COMBINE TYPE NOT IMPLEMENTED:

**Explanation:** The object file contains a segment combine type not implemented in LINK86. The object file is incompatible with LINK86, possibly because it was produced by an unsupported compiler or assembler, or because it was corrupted.

**System action:** LINK86 terminates and displays this error message.

**User response:** Correct the object file and restart LINK86 or use the correct linker for your compiler.

---

#### COMMAND TOO LONG:

**Explanation:** The total length of command input to LINK86, including the input file, exceeds the maximum of 2048 characters.

**System action:** LINK86 terminates and displays this error message.

**User response:** Reduce the command to within the maximum and restart LINK86.

---

#### DIRECTORY FULL:

**Explanation:** Not enough directory space exists for the output files.

**System action:** LINK86 terminates and displays this error message.

**User response:** Either erase some unnecessary files or get another disk with more directory space and run LINK86 again.

---

#### DISK READ ERROR:

**Explanation:** LINK86 cannot read an object or library file. This error is usually the result of an unexpected end-of-file.

**System action:** LINK86 terminates and displays this error message, followed by the name of the file.

**User response:** Replace the named file and try again.

---

#### DISK WRITE ERROR:

**Explanation:** A file cannot be written, probably because the disk is full.

**System action:** LINK86 terminates and displays this error message, followed by the name of the file it was trying to write.

**User response:** Use a disk with enough space for the file or take other appropriate action to make space available, and try again.

---

#### ERROR IN LIBATTR MODULE:

**Explanation:** The LIBATTR module in one of the libraries does not conform to established requirements.

**System action:** LINK86 terminates and displays this error message.

**User response:** Fix the LIBATTR module and rebuild the library in question. Then restart LINK86.

---

#### FIXUP TYPE NOT IMPLEMENTED:

**Explanation:** The object file uses a fixup type not implemented in LINK86. The object file is incompatible with LINK86, possibly because it was produced by an unsupported compiler or assembler, or because it was corrupted.

**System action:** LINK86 terminates and displays this error message.

**User response:** Correct the object file and restart LINK86, or use the correct linker for your compiler.

---

#### GROUP NOT FOUND:

**Explanation:** The group name specified in the command line does not appear in any of the files linked.

**System action:** LINK86 terminates and displays this error message, followed by the name of the missing group.

**User response:** Enter the correct group name in the command line, or change the group name in the files being linked and try again.

## Error Messages

---

### GROUP OVER 64 KB:

**Explanation:** The group listed is larger than 64K.

**System action:** LINK86 terminates and displays this error message, followed by the name of the group.

**User response:** Delete segments from the group, divide it into two or more groups, or do not use groups. Then restart LINK86.

---

### GROUP TYPE NOT IMPLEMENTED:

**Explanation:** The object file contains a segment that is not an element of any known group.

**System action:** LINK86 terminates and displays this error message, followed by the name of the segment.

**User response:** Correct the object file and restart LINK86.

---

### INVALID LIBRARY-REQUESTED SUFFIX:

**Explanation:** The load module suffix requested by the LIBATTR module in a library is not supported.

**System action:** LINK86 terminates and displays this error message, followed by the load module suffix.

**User response:** Correct the LIBATTR module within the library and restart LINK86.

---

### LINK86 ERROR 1:

**Explanation:** An inconsistency exists in the LINK86 internal tables.

**System action:** LINK86 terminates and displays this error message.

**User response:** If an internal linker error occurs during the linking of your application program, copy your object files, runtime libraries, and input file onto a diskette. Then relink your application with the same command line options, but add the following text to the end of the command line, *preceded by a single blank space*:

```
>filename.ext
```

This command creates a file that captures the console output of the linker. Copy filename.ext onto the same diskette as the files and libraries. Contact your service representative when you have gathered this information.

---

### MORE THAN ONE MAIN PROGRAM:

**Explanation:** The source program has more than one main program.

**System action:** LINK86 terminates and displays this error message.

**User response:** Correct the source program, recompile, and restart LINK86.

---

### MULTIPLE DEFINITION:

**Explanation:** The indicated symbol is defined as PUBLIC in more than one module.

**System action:** LINK86 terminates and displays this error message, followed by the name of the symbol.

**User response:** Correct the problem in the source files, regenerate the object files, and try again.

---

### NO FILE:

**Explanation:** LINK86 cannot find the indicated input, object, or library on the indicated drive.

**System action:** LINK86 terminates and displays this error message, followed by the name of the missing file.

**User response:** Correct the file name or drive identifier and try again.

---

### OBJECT FILE ERROR *nn*:

**Explanation:** LINK86 detected an error in the object file. This error is caused by a compiler error or by a bad disk file. The error codes (*nn*) are defined in "Object File Error Codes" on page 738.

**System action:** LINK86 terminates and displays this error message, followed by the file name, record number, and item number.

**User response:** Regenerate the object file and link again. If the error still occurs, copy the source file used to create the object file and copy both files onto a diskette. Then relink your application with the same command line options, but add the following to the end of the command line, *preceded by a single blank space*:

```
>filename.ext
```

This creates a file that captures the console output of the linker. Copy filename.ext onto the same diskette as the files and libraries. Contact your service representative when you have gathered this information.

---

### RECORD TYPE NOT IMPLEMENTED:

**Explanation:** The object file contains a record type not implemented in LINK86. The object file is incompatible with LINK86, possibly because it was produced by an unsupported compiler or assembler, or because it was corrupted.

**System action:** LINK86 terminates and displays this error message, followed by the name of the record type.

**User response:** Correct the object file and restart LINK86, or use the correct linker for your compiler.



---

**SEGMENT ATTRIBUTE ERROR:**

**Explanation:** The combine type of the indicated segment is not the same as the type of segment in a previously linked file.

**System action:** LINK86 terminates and displays this error message, followed by the name of the segment.

**User response:** Change the segment attributes in your source file as needed, regenerate the object file, and then link again.

---

**SEGMENT CLASS ERROR:**

**Explanation:** The class of a segment is not one of the following: CODE, DATA, STACK, EXTRA, X1, X2, X3, or X4.

**System action:** LINK86 terminates and displays this error message, followed by the name of the segment.

**User response:** Correct the object file and restart LINK86.

---

**SEGMENT COMBINATION ERROR:**

**Explanation:** An attempt is made to combine segments that cannot be combined, such as LOCAL segments.

**System action:** LINK86 terminates and displays this error message.

**User response:** Change the segment attributes in your source file, regenerate the object file, and relink.

---

**SEGMENT NOT FOUND:**

**Explanation:** The segment name specified in the command line does not appear in any of the files linked.

**System action:** LINK86 terminates and displays this error message, followed by the name of the segment.

**User response:** Enter the correct segment name in the command line or change the segments names in the files being linked, and try again.

---

**SEGMENT OVER 64K:**

**Explanation:** The indicated segment has a total length greater than 64 KB.

**System action:** LINK86 terminates and displays this error message, followed by the name of the segment.

**User response:** Reduce the segment's size, or do not combine it with PUBLIC segments that have the same name. If the CODE segment exceeds 64K, be sure that all libraries (.L86 files) are linked using the SEARCH option to minimize code size.

---

**SRTL DATA OVERLAP:**

**Explanation:** The data from two SRTLs overlaps.

**System action:** LINK86 terminates and displays this error message, followed by the name of the SRTL.

**User response:** Change the base address in the LIBATTR module of one of the SRTLs, and restart LINK86.

---

**STACK COLLIDES WITH SRTL DATA:**

**Explanation:** The base address of SRTL data does not allow enough room for the requested amount of stack space.

**System action:** LINK86 terminates and displays this error message, followed by the name of the SRTL.

**User response:** Change the base of the SRTL data in the LIBATTR module, or request less stack space. Then restart LINK86.

---

**SYMBOL TABLE OVERFLOW:**

**Explanation:** LINK86 ran out of symbol table space.

**System action:** LINK86 terminates and displays this error message.

**User response:** Either reduce the number or length of PUBLIC and global symbols in the program, or, if there is not enough memory available for the linker to obtain a 64K segment, relink on a system with more memory available.

---

**SYNTAX ERROR:**

**Explanation:** LINK86 detected a syntax error in the command line, probably because of an improper file name or an command option that is not valid.

**System action:** LINK86 stops, displays this error message, and echoes the command line up to the point where it found the error.

**User response:** Retype the command line or edit the INP file, and try again.

---

**TARGET OUT OF RANGE:**

**Explanation:** The target of a fixup cannot be reached from the location of the fixup. This usually means the fixup and target are more than 64K bytes apart.

**System action:** LINK86 terminates and displays this error message.

**User response:** Correct your source files, regenerate the object files, and try again.

## Error Messages

---

### TOO MANY MODULES IN LIBRARY:

**Explanation:** The library contains more modules than LINK86 can handle.

**System action:** LINK86 terminates and displays this error message, followed by the name of the library.

**User response:** Split the library into two or more libraries and restart LINK86.

---

### TOO MANY MODULES LINKED FROM LIBRARY:

**Explanation:** A library is supplying more than 256 modules during a single execution of LINK86.

**System action:** LINK86 terminates and displays this error message, followed by the name of the library.

**User response:** Split the library into two or more libraries and restart LINK86.

---

### UNDEFINED SYMBOLS:

**Explanation:** The symbols following this message are referenced but not defined in any of the modules being linked.

**System action:** LINK86 terminates and displays this

error message, followed by the undefined symbols.

**User response:** Edit your source files to define the symbols in the modules being linked, regenerate the object files, and try again.

---

### XSRTL MUST BE LINKED BY ITSELF:

**Explanation:** Other files are being linked at the same time as an executable shared runtime library (XSRTL).

**System action:** LINK86 terminates and displays this error message.

**User response:** Run LINK86 on the XSRTL alone.

---

### XSRTLs INCOMPATIBLE WITH OVERLAYS:

**Explanation:** The application program being linked with the executable shared runtime library contains overlays.

**System action:** LINK86 terminates and displays this error message, followed by the name of the XSRTL.

**User response:** Remove the overlays and restart LINK86 or specify the NOSHARED option for the runtime library and restart LINK86.

## Object File Error Codes

The following error codes appear only when LINK86 encounters an object file or library file that does not conform to the Intel 8086 Object Module Format specification.

### Error Code

#### Description

- |    |   |
|----|---|
| 1  | File name in COMMENT record has more than eight characters. |
| 2  | LEDATA record has more than 1024 bytes of data.             |
| 3  | Data in LEDATA/LIDATA record extends past end of segment.   |
| 4  | Checksum error.   |
| 5  | Record type is not valid.                                   |
| 6  | Library record found where not expected.                    |
| 7  | No library record found where expected.                     |
| 8  | Index out of range.   |
| 9  | File or module does not start with THEADR or LIBHED record. |
| 10 | Name too long (more than 40 characters).                    |
| 11 | Character in name is not valid.                             |
| 12 | Record length is not valid.                                 |
| 13 | Repeat count = 0 in LIDATA record.                          |
| 14 | Error expanding LIDATA record.                              |
| 15 | Bad field in MODEND record (MODTYP L=0).                    |
| 16 | Bad fixup type in MODEND record.                            |



- 20** Target thread (method field > 3) is not valid.
- 21** Frame thread (method field > 6) is not valid.
- 22** Loc field (lof > 4) is not valid.
- 23** Frame field in FIXDAT (frame > 3 when thread fixup) is not valid.
- 24** Reference to nonexistent frame thread.
- 25** Frame field in FIXDAT (frame > 6 when explicit fixup) is not valid.
- 26** Reference to nonexistent target thread.
- 27** Self-relative fixup (not low byte or offset byte) is not valid.
- 28** Fixup goes past LEDATA record size.
- 29** Cannot reach start address specified in MODEND record.

## Error Messages

---

## Appendix C. Character Sets and Check Printing Application

This appendix contains the normal and double-width character sets used to print checks. It also contains an example application that was tested for check printing. For more information, see “Printing Checks” on page 101.

---

### Example of a Normal Width Character Set

The following is an example of the character set used in check printing. It contains normal width characters (5 x 8). The values are in hexadecimal.

00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00	SP
1F,00,00,00,1F,00,00,00,1F,00,00,00,1F,00,00,00	!
0A,00,0A,00,0A,00,00,00,00,00,00,00,00,00,00,00	"
00,00,1F,00,00,00,1F,00,00,00,1F,00,00,00,00,00	#
00,00,00,1F,00,00,00,1F,00,00,00,1F,00,00,00,00	\$
1F,00,00,00,1F,00,00,00,1F,00,00,00,1F,00,00,00	%
04,00,0A,00,00,0A,04,00,08,00,15,00,12,00,0D,00	&
04,00,04,00,04,00,00,00,00,00,00,00,00,00,00,00	'
02,04,08,00,08,00,00,10,00,00,08,00,08,04,02,00	(
08,04,02,00,02,00,00,01,00,00,02,00,02,04,08,00	)
00,00,11,00,0A,00,1F,00,0A,00,11,00,00,00,00,00	*
00,00,00,00,00,00,0C,00,0C,00,04,08,10,00,00,00	,
00,00,00,00,00,00,1F,00,00,00,00,00,00,00,00,00	-
00,00,00,00,00,00,00,00,00,00,00,00,0C,00,0C,00	.
01,00,02,00,02,00,04,00,04,00,08,00,08,00,10,00	/
0E,00,11,00,11,02,11,04,11,08,11,00,11,00,0E,00	0
04,00,0C,00,04,00,04,00,04,00,04,00,04,00,0E,00	1
0E,00,11,00,02,00,04,00,08,00,10,00,10,00,1F,00	2
1E,00,01,00,01,00,07,00,01,00,01,00,01,00,1E,00	3
11,00,11,00,11,00,11,00,1F,00,01,00,01,00,01,00	4
1F,00,10,00,10,00,1E,00,01,00,01,00,01,00,1E,00	5
0E,00,11,00,10,00,1E,00,11,00,11,00,11,00,0E,00	6
1F,00,01,00,02,00,04,00,08,00,10,00,10,00,10,00	7
0E,00,11,00,11,00,0E,00,11,00,11,00,11,00,0E,00	8
0E,00,11,00,11,00,0F,00,01,00,01,00,01,00,1E,00	9
02,00,04,00,08,00,10,00,10,00,08,00,04,00,02,00	<
08,00,04,00,02,00,01,00,01,00,02,00,04,00,08,00	>
04,00,0A,00,11,00,11,00,1F,00,11,00,11,00,11,00	A
1E,00,11,00,11,00,1E,00,11,00,11,00,11,00,1E,00	B
0E,00,11,00,10,00,10,00,10,00,10,00,11,00,0E,00	C
1C,02,11,00,11,00,11,00,11,00,11,00,11,02,1C,00	D
1F,00,10,00,10,00,1C,00,10,00,10,00,10,00,1F,00	E
1F,00,10,00,10,00,1C,00,10,00,10,00,10,00,10,00	F
0E,00,11,00,10,00,10,00,17,00,11,00,11,00,0E,00	G
11,00,11,00,11,00,1F,00,11,00,11,00,11,00,11,00	H
0E,00,04,00,04,00,04,00,04,00,04,00,04,00,0E,00	I
02,00,02,00,02,00,02,00,02,00,02,00,12,00,0C,00	J
11,00,12,00,14,00,18,00,18,00,14,00,12,00,11,00	K
10,00,10,00,10,00,10,00,10,00,10,00,10,00,1F,00	L
11,00,1B,00,15,00,15,00,11,00,11,00,11,00,11,00	M
11,00,19,00,19,00,15,00,13,00,13,00,11,00,11,00	N
0E,00,11,00,11,00,11,00,11,00,11,00,11,00,0E,00	O
1E,00,11,00,11,00,1E,00,10,00,10,00,10,00,10,00	P
0E,00,11,00,11,00,11,00,11,00,15,00,13,00,0F,00	Q
1E,00,11,00,11,00,1E,00,18,00,14,00,12,00,11,00	R
0E,00,11,00,10,00,0E,00,01,00,01,00,11,00,0E,00	S
1F,00,04,00,04,00,04,00,04,00,04,00,04,00,04,00	T
11,00,11,00,11,00,11,00,11,00,11,00,11,00,0E,00	U
11,00,11,00,11,00,11,00,11,00,11,00,0A,00,04,00	V

11,00,11,00,11,00,11,00,15,00,15,00,1B,00,11,00	W
11,00,11,00,0A,00,04,00,04,00,0A,00,11,00,11,00	X
11,00,11,00,0A,00,04,00,04,00,04,00,04,00,04,00	Y
1F,00,00,01,00,02,00,04,00,08,00,10,00,00,1F,00	Z

---

## Example of a Double Width Character Set

This example contains double-width characters (8 x 8). The values are in hexadecimal.

7E,00,83,00,85,00,89,00,91,00,A1,00,C1,00,7E,00	0
18,00,28,00,08,00,08,00,08,00,08,00,08,00,3E,00	1
7E,00,81,00,01,02,04,08,10,20,40,00,80,00,FF,00	2
FC,02,01,00,01,02,0C,02,01,00,01,00,01,02,FC,00	3
41,00,41,00,41,00,41,00,7F,00,01,00,01,00,01,00	4
FE,00,80,00,80,00,FC,02,01,00,01,00,01,02,7C,00	5
6E,40,80,00,80,00,FC,02,81,00,81,00,81,42,3C,00	6
FF,00,01,02,00,04,00,08,00,10,20,00,20,00,00,20	7
3C,42,81,00,81,42,3C,42,81,00,81,00,81,42,3C,00	8
3C,42,81,00,81,00,7F,00,01,00,01,00,01,02,7C,00	9
7E,00,54,00,2A,00,54,00,2A,00,54,00,2A,00,7E,00	Box

---

## Example Application for Printing Checks on the Model 2 Printer

The following example demonstrates the APACS Check Printing Capability of the Model 2 printer. It prints a check on the document insert station.

The application begins by opening the document insert station and performing a PUTLONG. This statement begins the check printing mode. Then the paper advances to the amount field and the amount is printed in double-width characters. This process takes ten print passes including one last blank print pass. The remainder of the check data is printed in 36 passes. Check printing normally takes a maximum of 9.5 seconds.

The example application includes the check data in the program as data statements. Your application would not use data statements. The data would be read in from a lookup table or a disk file. This application has minimal error recovery.

```
%ENVIRON T
INTEGER*2 I%
INTEGER*2 J%
INTEGER*4 X%
INTEGER*1 TEMP%(1)
DIM TEMP%(381)
SUB ASYNCSUB(RFLAG,OVER)
  INTEGER*2 RFLAG
  STRING OVER
  WAIT; 10000
  ! Display the error.
  CLEARS 10
  WRITE #10; "ASYNC ERROR = ",ERR
  ! Do not retry.
  RFLAG = 0
  EXIT SUB
END SUB
ON ASYNC ERROR CALL ASYNCSUB
ON ERROR GOTO ERR.HNDLR

! Load the input state table. This is a table
! that sets up a motor key.
LOAD "ISTBL = R::ADX_UPGM:APAC0.TBL"
! Open the DI Station
```

```

OPEN "DI:" AS 5
! Open the display with a greeting.
OPEN "ANDISPLAY:" AS 10
CLEARS 10
WRITE #10; "APACS CHECK PRINTING"
WRITE #10; "DEMO"
! Leave message up five seconds.
WAIT; 5000
! Open the I/O Processor and unlock to the initial state.
OPEN "IOPROC:" AS 11
UNLOCKDEV 11, 1
! Put the printer in check printing mode and auto-insert mode.
Putlong 5, 00000078H
! Open Loop
WHILE 1
! Wait to press the ENTER key.
    READ #11; LINE A$
    LOCKDEV 11, PURGE
! Advance the paper to the amount field.
    WRITE FORM "C38 A9";#5; "
! Print field starts at 94 primary positions to the left of the
! home position, and continues 72 print positions.
    TEMP%(1) = 94
    TEMP%(2) = 72
! Print the amount field using double wide characters.
! It is necessary to advance the paper 9 steps after
! every print pass. Because each print pass has 72 bytes,
! there are (380 - 3 control bytes) 377 bytes to send print
! data. 5 print passes can be written at a time. The amount field
! consists of a total of 10 passes, so it can do this in two writes.
! The first pass is blank to prevent incomplete line feeding
! on the first pass.
    TEMP%(3) = 5
    TEMP%(381) = 9
    FOR J%=1 TO 2
        FOR I%=4 TO 363
! 72 Bytes per print pass,
! 5 print passes per write
            READ TEMP%(I%)
            NEXT I%
            WRITE LOGO #5; TEMP%(1)
        NEXT J%
! Print the rest of the check. This part of the check consists
! of normal sized characters (5x8 font). The paper should be
! advanced 6 steps after printing each pass. On the last print
! pass, turn on the high order bit of byte 381. This tells the
! driver to return the print head to home position after printing
! the last line of the WRITE LOGO. A TCLOSE can also return
! the print head to home position, but it forces the application
! to wait until all queued prints are done.
        TEMP%(381) = 6
        FOR J%=1 TO 7
            FOR I%=4 TO 363
! 72 bytes per print pass, 5 print passes per write
                READ TEMP%(I%)
                NEXT I%
                WRITE LOGO #5; TEMP%(1)
            NEXT J%
! Do only one pass this time.
            TEMP%(3) = 1
! High order bit of line feed byte to home head set to ON.
            TEMP%(381) = 86H
            FOR I%=4 TO 75
                READ TEMP%(I%)
            NEXT I%
            WRITE LOGO #5; TEMP%(1)
! Allow keyboard input

```

```

        UNLOCKDEV 11, 1
! Reposition the data statement pointer to the beginning
        RESTORE
WEND
ERR.HNDLR
! Display the error.
WAIT; 10000
CLEAR$ 10
WRITE #10; "SYNC ERROR = ",ERR
RESUME
! Passes 1-10 are for the double printing sections of the check.
! Pass 1 - blank
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0
! Pass 2 - "box" character
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 126, 0, 84, 0, 42, 0, 84, 0
DATA 42, 0, 84, 0, 42, 0, 126, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0
! Pass 3 - "4"
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 65, 0, 65, 0, 65, 0, 65, 0
DATA 127, 0, 1, 0, 1, 0, 1, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0
! Pass 4 - "2"
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 126, 0, -127, 0, 1, 2, 4, 8
DATA 16, 32, 64, 0, -128, 0, -1, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0
! Pass 5 - "-"
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 31
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0
! Pass 6 - "8"
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 60, 66, -127, 0, -127, 66, 60, 66
DATA -127, 0, -127, 0, -127, 66, 60, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0
! Pass 7 - "7"
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

```

```

DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, -1, 0, 1, 2, 0, 4, 0, 8
DATA 0, 16, 32, 0, 32, 0, 0, 32, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0

! Pass 8 - "3"
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, -4, 2, 1, 0, 1, 2, 12, 2
DATA 1, 0, 1, 0, 1, 2, -4, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0

! Pass 9 - "box" character
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 126, 0, 84, 0, 42, 0, 84, 0
DATA 42, 0, 84, 0, 42, 0, 126, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0

! Pass 10 - blank
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0

! This is the normal width part of the check.
! Pass 11
DATA 14, 0, 17, 0, 17, 0, 14, 0, 17, 0
DATA 17, 0, 17, 0, 14, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0

! Pass 12
DATA 14, 0, 17, 0, 17, 0, 14, 0, 17, 0
DATA 17, 0, 17, 0, 14, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0

! Pass 13
DATA 30, 0, 17, 0, 17, 0, 30, 0, 24, 0
DATA 20, 0, 18, 0, 17, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0

! Pass 14
DATA 4, 0, 10, 0, 17, 0, 17, 0, 31, 0
DATA 17, 0, 17, 0, 17, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

```

```

DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0
! Pass 15
DATA 17, 0, 27, 0, 21, 0, 21, 0, 17, 0
DATA 17, 0, 17, 0, 17, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0
! Pass 16
DATA 14, 0, 17, 0, 16, 0, 30, 0, 17, 0
DATA 17, 0, 17, 0, 14, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0
! Pass 17
DATA 4, 0, 12, 0, 4, 0, 4, 0, 4, 0
DATA 4, 0, 4, 0, 14, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0
! Pass 18
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 17, 0, 10, 0, 31, 0
DATA 10, 0, 17, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0
! Pass 19
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 17, 0, 10, 0, 31, 0
DATA 10, 0, 17, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0
! Pass 20
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 31, 0, 4, 0, 4, 0, 4, 0
DATA 4, 0, 4, 0, 4, 0, 4, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0
! Pass 21
DATA 30, 0, 17, 0, 17, 0, 30, 0, 24, 0
DATA 20, 0, 18, 0, 17, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 17, 0, 17, 0, 17, 0, 31, 0
DATA 17, 0, 17, 0, 17, 0, 17, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

```



```

DATA 0, 0
! Pass 22
DATA 31, 0, 16, 0, 16, 0, 28, 0, 16, 0
DATA 16, 0, 16, 0, 31, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 14, 0, 17, 0, 16, 0, 16, 0
DATA 23, 0, 17, 0, 17, 0, 14, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0
! Pass 23
DATA 14, 0, 17, 0, 16, 0, 14, 0, 1, 0
DATA 1, 0, 17, 0, 14, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 14, 0, 4, 0, 4, 0, 4, 0
DATA 4, 0, 4, 0, 4, 0, 14, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0
! Pass 24
DATA 17, 0, 17, 0, 17, 0, 17, 0, 17, 0
DATA 17, 0, 17, 0, 14, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 31, 0, 16, 0, 16, 0, 28, 0
DATA 16, 0, 16, 0, 16, 0, 31, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0
! Pass 25
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 17, 0, 17, 0, 10, 0, 4, 0
DATA 4, 0, 4, 0, 4, 0, 4, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0
! Pass 26
DATA 17, 0, 27, 0, 21, 0, 21, 0, 17, 0
DATA 17, 0, 17, 0, 17, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 31, 0, 4, 0, 4, 0, 4, 0
DATA 4, 0, 4, 0, 4, 0, 4, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0
! Pass 27
DATA 31, 0, 16, 0, 16, 0, 28, 0, 16, 0
DATA 16, 0, 16, 0, 31, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 17, 0, 25, 0, 25, 0, 21, 0
DATA 19, 0, 19, 0, 17, 0, 17, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0
! Pass 28
DATA 31, 0, 4, 0, 4, 0, 4, 0, 4, 0
DATA 4, 0, 4, 0, 4, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 31, 0, 16, 0, 16, 0, 28, 0
DATA 16, 0, 16, 0, 16, 0, 31, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0
! Pass 29
DATA 14, 0, 17, 0, 16, 0, 14, 0, 1, 0

```

```

DATA 1, 0, 17, 0, 14, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 17, 0, 17, 0, 17, 0, 17, 0
DATA 17, 0, 17, 0, 10, 0, 4, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0
! Pass 30
DATA 17, 0, 17, 0, 10, 0, 4, 0, 4, 0
DATA 4, 0, 4, 0, 4, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 31, 0, 16, 0, 16, 0, 28, 0
DATA 16, 0, 16, 0, 16, 0, 31, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0
! Pass 31
DATA 14, 0, 17, 0, 16, 0, 14, 0, 1, 0
DATA 1, 0, 17, 0, 14, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 14, 0, 17, 0, 16, 0, 14, 0
DATA 1, 0, 1, 0, 17, 0, 14, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0
! Pass 32
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 4, 0, 10, 0, 0, 10, 4, 0
DATA 8, 0, 21, 0, 18, 0, 13, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0
! Pass 33
DATA 31, 0, 16, 0, 16, 0, 28, 0, 16, 0
DATA 16, 0, 16, 0, 31, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 28, 2, 17, 0, 17, 0, 17, 0
DATA 17, 0, 17, 0, 17, 2, 28, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0
! Pass 34
DATA 30, 0, 17, 0, 17, 0, 30, 0, 24, 0
DATA 20, 0, 18, 0, 17, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 31, 0, 16, 0, 16, 0, 28, 0
DATA 16, 0, 16, 0, 16, 0, 31, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 17, 0
DATA 10, 0, 31, 0, 10, 0, 17, 0, 0, 0
DATA 0, 0
! Pass 35
DATA 14, 0, 17, 0, 17, 0, 17, 0, 17, 0
DATA 17, 0, 17, 0, 14, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 30, 0, 17, 0, 17, 0, 30, 0
DATA 24, 0, 20, 0, 18, 0, 17, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 17, 0
DATA 10, 0, 31, 0, 10, 0, 17, 0, 0, 0
DATA 0, 0
! Pass 36
DATA 31, 0, 4, 0, 4, 0, 4, 0, 4, 0
DATA 4, 0, 4, 0, 4, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 28, 2, 17, 0, 17, 0, 17, 0

```

DATA	17,	0,	17,	0,	17,	2,	28,	0,	0,	0
DATA	0,	0,	0,	0,	0,	0,	30,	0,	17,	0
DATA	17,	0,	30,	0,	16,	0,	16,	0,	16,	0
DATA	16,	0								
! Pass 37										
DATA	14,	0,	17,	0,	16,	0,	14,	0,	1,	0
DATA	1,	0,	17,	0,	14,	0,	0,	0,	0,	0
DATA	0,	0,	0,	0,	0,	0,	0,	0,	0,	0
DATA	0,	0,	17,	0,	25,	0,	25,	0,	21,	0
DATA	19,	0,	17,	0,	17,	0,	17,	0,	0,	0
DATA	0,	0,	0,	0,	0,	0,	17,	0,	17,	0
DATA	17,	0,	17,	0,	31,	0,	1,	0,	1,	0
DATA	1,	0								
! Pass 38										
DATA	0,	0,	0,	0,	0,	0,	0,	0,	0,	0
DATA	0,	0,	0,	0,	0,	0,	0,	0,	0,	0
DATA	0,	0,	0,	0,	0,	0,	0,	0,	0,	0
DATA	0,	0,	17,	0,	17,	0,	17,	0,	17,	0
DATA	17,	0,	17,	0,	17,	0,	14,	0,	0,	0
DATA	0,	0,	0,	0,	0,	0,	14,	0,	17,	0
DATA	2,	0,	4,	0,	8,	0,	16,	0,	16,	0
DATA	31,	0								
! Pass 39										
DATA	14,	0,	17,	0,	17,	2,	17,	4,	17,	8
DATA	17,	0,	17,	0,	14,	0,	0,	0,	0,	0
DATA	0,	0,	0,	0,	0,	0,	0,	0,	0,	0
DATA	0,	0,	17,	0,	17,	0,	17,	0,	31,	0
DATA	17,	0,	17,	0,	17,	0,	17,	0,	0,	0
DATA	0,	0,	0,	0,	0,	0,	14,	0,	17,	0
DATA	16,	0,	14,	0,	1,	0,	1,	0,	17,	0
DATA	14,	0								
! Pass 40										
DATA	14,	0,	17,	0,	17,	0,	14,	0,	17,	0
DATA	17,	0,	17,	0,	14,	0,	0,	0,	0,	0
DATA	0,	0,	0,	0,	0,	0,	0,	0,	0,	0
DATA	0,	0,	31,	0,	16,	0,	16,	0,	28,	0
DATA	16,	0,	16,	0,	16,	0,	31,	0,	0,	0
DATA	0,	0,	0,	0,	0,	0,	28,	2,	17,	0
DATA	17,	0,	17,	0,	17,	0,	17,	0,	17,	2
DATA	28,	0								
! Pass 41										
DATA	14,	0,	17,	0,	16,	0,	30,	0,	17,	0
DATA	17,	0,	17,	0,	14,	0,	0,	0,	0,	0
DATA	0,	0,	0,	0,	0,	0,	0,	0,	0,	0
DATA	0,	0,	31,	0,	16,	0,	16,	0,	28,	0
DATA	16,	0,	16,	0,	16,	0,	31,	0,	0,	0
DATA	0,	0,	0,	0,	0,	0,	17,	0,	25,	0
DATA	25,	0,	21,	0,	19,	0,	19,	0,	17,	0
DATA	17,	0								
! Pass 42										
DATA	17,	0,	17,	0,	17,	0,	17,	0,	31,	0
DATA	1,	0,	1,	0,	1,	0,	0,	0,	0,	0
DATA	0,	0,	0,	0,	0,	0,	0,	0,	0,	0
DATA	0,	0,	30,	0,	17,	0,	17,	0,	30,	0
DATA	24,	0,	20,	0,	18,	0,	17,	0,	0,	0
DATA	0,	0,	0,	0,	0,	0,	17,	0,	17,	0
DATA	17,	0,	17,	0,	17,	0,	17,	0,	17,	0
DATA	14,	0								
! Pass 43										
DATA	0,	0,	0,	0,	0,	0,	0,	0,	0,	0
DATA	0,	0,	0,	0,	0,	0,	0,	0,	0,	0
DATA	0,	0,	0,	0,	0,	0,	0,	0,	0,	0
DATA	0,	0,	17,	0,	17,	0,	17,	0,	31,	0
DATA	17,	0,	17,	0,	17,	0,	17,	0,	0,	0
DATA	0,	0,	0,	0,	0,	0,	14,	0,	17,	0
DATA	17,	0,	17,	0,	17,	0,	17,	0,	17,	0

```

DATA 14, 0
! Pass 44
DATA 17, 0, 27, 0, 21, 0, 21, 0, 17, 0
DATA 17, 0, 17, 0, 17, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 31, 0, 4, 0, 4, 0, 4, 0
DATA 4, 0, 4, 0, 4, 0, 4, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 30, 0, 17, 0
DATA 17, 0, 30, 0, 16, 0, 16, 0, 16, 0
DATA 16, 0
! Pass 45
DATA 30, 0, 17, 0, 17, 0, 30, 0, 17, 0
DATA 17, 0, 17, 0, 30, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 17, 0, 10, 0, 31, 0
DATA 10, 0, 17, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 17, 0
DATA 10, 0, 31, 0, 10, 0, 17, 0, 0, 0
DATA 0, 0
! Pass 46
DATA 14, 0, 4, 0, 4, 0, 4, 0, 4, 0
DATA 4, 0, 4, 0, 14, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 17, 0, 10, 0, 31, 0
DATA 10, 0, 17, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 17, 0
DATA 10, 0, 31, 0, 10, 0, 17, 0, 0, 0
DATA 0, 0

```

---

## Appendix D. Notices

This information was developed for products and services offered in the U.S.A.

Toshiba Global Commerce Solutions may not offer the products, services, or features discussed in this document in other countries. Consult your local Toshiba Global Commerce Solutions representative for information on the products and services currently available in your area. Any reference to a Toshiba Global Commerce Solutions product, program, or service is not intended to state or imply that only that Toshiba Global Commerce Solutions product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any Toshiba Global Commerce Solutions intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-Toshiba Global Commerce Solutions product, program, or service.

Toshiba Global Commerce Solutions may have patents or pending patent applications covering the subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

Toshiba Global Commerce Solutions  
Attn: General Counsel  
3039 E. Cornwallis Rd  
RTP, NC 27709

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: TOSHIBA GLOBAL COMMERCE SOLUTIONS PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. Toshiba Global Commerce Solutions may make improvements and/or changes in the product(s) and/or program(s) described in this publication at any time without notice.

Toshiba Global Commerce Solutions may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Any references in this information to non-Toshiba Global Commerce Solutions Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this Toshiba Global Commerce Solutions product and use of those Web sites is at your own risk.

Information concerning non-Toshiba Global Commerce Solutions products was obtained from the suppliers of those products, their published announcements or other publicly available sources. Toshiba Global Commerce Solutions has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-Toshiba Global Commerce Solutions products. Questions on the capabilities of non-Toshiba Global Commerce Solutions products should be addressed to the suppliers of those products.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

---

## Telecommunication regulatory statement

This product is not intended to be connected directly or indirectly by any means whatsoever to interfaces of public telecommunications networks, nor is it intended to be used in a public services network.

---

## Electronic Emission Notices

When you attach a monitor to the equipment, you must use the designated monitor cable and any interference suppression devices that are supplied with the monitor.

## Federal Communications Commission (FCC) Statement

This equipment has been tested and found to comply with the limits for a Class A digital device, pursuant to Part 15 of the FCC Rules. These limits are designed to provide reasonable protection against harmful interference when the equipment is operated in a commercial environment. This equipment generates, uses, and can radiate radio frequency energy and, if not installed and used in accordance with the instruction manual, may cause harmful interference to radio communications. Operation of this equipment in a residential area is likely to cause harmful interference, in which case the user will be required to correct the interference at his own expense.

Properly shielded and grounded cables and connectors must be used in order to meet FCC emission limits. Toshiba Global Commerce Solutions is not responsible for any radio or television interference caused by using other than recommended cables and connectors or by unauthorized changes or modifications to this equipment. Unauthorized changes or modifications could void the user's authority to operate the equipment.

This device complies with part 15 of the FCC Rules. Operation is subject to the following two conditions:

1. This device may not cause harmful interference, and
2. This device must accept any interference received, including interference that may cause undesired operation.

## Industry Canada Class A Emission Compliance statement

This Class A digital apparatus complies with Canadian ICES-003.

## Avis de conformité à la réglementation d'Industrie Canada

Cet appareil numérique de la classe A est conforme à la norme NMB-003 du Canada.

## Australia and New Zealand Class A Statement

**Attention:** This is a Class A product. In a domestic environment this product may cause radio interference, in which case the user may be required to take adequate measures.

## European Union Electromagnetic Compatibility (EMC) Directive Conformance Statement

This product is in conformity with the protection requirements of EU Council Directive 2004/108/EC on the approximation of the laws of the Member States relating to electromagnetic compatibility. Toshiba Global Commerce Solutions cannot accept responsibility for any failure to satisfy the protection requirements resulting from a non-recommended modification of the product, including the fitting of non-Toshiba Global Commerce Solutions option cards.

This product has been tested and found to comply with the limits for Class A Information Technology Equipment according to CISPR 22/European Standard EN 55022. The limits for Class A equipment were derived for commercial and industrial environments to provide reasonable protection against interference with licensed communication equipment.

**Attention:** This is a Class A product. In a domestic environment this product may cause radio interference in which case the user may be required to take adequate measures.

Responsible manufacturer:

Toshiba Global Commerce Solutions  
3039 Cornwallis Road  
Building 307  
Research Triangle Park, North Carolina 27709  
United States of America

European Community contact:

Toshiba Global Commerce Solutions  
Brand Manager - Europe, Middle East & Africa  
3 NEW SQUARE, FELTHAM, TW14 8HB Great Britain  
Building: 1 Floor: NA | Office: MOBILE  
Tel: 44-7967-275819  
e-mail: robin\_lyon@uk.ibm.com

## **Germany Class A Statement**

### **Deutschsprachiger EU Hinweis: Hinweis für Geräte der Klasse A EU-Richtlinie zur Elektromagnetischen Verträglichkeit**

Dieses Produkt entspricht den Schutzanforderungen der EU-Richtlinie 2004/108/EG zur Angleichung der Rechtsvorschriften über die elektromagnetische Verträglichkeit in den EU-Mitgliedsstaaten und hält die Grenzwerte der EN 55022 Klasse A ein.

Um dieses sicherzustellen, sind die Geräte wie in den Handbüchern beschrieben zu installieren und zu betreiben. Des Weiteren dürfen auch nur von der Toshiba Global Commerce Solutions empfohlene Kabel angeschlossen werden. Toshiba Global Commerce Solutions übernimmt keine Verantwortung für die Einhaltung der Schutzanforderungen, wenn das Produkt ohne Zustimmung der Toshiba Global Commerce Solutions verändert bzw. wenn Erweiterungskomponenten von Fremdherstellern ohne Empfehlung der Toshiba Global Commerce Solutions gesteckt/eingebaut werden.

EN 55022 Klasse A Geräte müssen mit folgendem Warnhinweis versehen werden: "Warnung: Dieses ist eine Einrichtung der Klasse A. Diese Einrichtung kann im Wohnbereich Funk-Störungen verursachen; in diesem Fall kann vom Betreiber verlangt werden, angemessene Maßnahmen zu ergreifen und dafür aufzukommen."

### **Deutschland: Einhaltung des Gesetzes über die elektromagnetische Verträglichkeit von Geräten**

Dieses Produkt entspricht dem "Gesetz über die elektromagnetische Verträglichkeit von Geräten (EMVG)". Dies ist die Umsetzung der EU-Richtlinie 2004/108/EG in der Bundesrepublik Deutschland.

### **Zulassungsbescheinigung laut dem Deutschen Gesetz über die elektromagnetische Verträglichkeit von Geräten (EMVG) (bzw. der EMC EG Richtlinie 2004/108/EG) für Geräte der Klasse A**

Dieses Gerät ist berechtigt, in Übereinstimmung mit dem Deutschen EMVG das EG-Konformitätszeichen - CE - zu führen.

Verantwortlich für die Einhaltung der EMV Vorschriften ist der Hersteller:

Toshiba Global Commerce Solutions  
3039 Cornwallis Road  
Building 307  
Research Triangle Park, North Carolina 27709  
United States of America

Der verantwortliche Ansprechpartner des Herstellers in der EU ist:

Toshiba Global Commerce Solutions  
Brand Manager - Europe, Middle East & Africa  
3 NEW SQUARE, FELTHAM, TW14 8HB Great Britain  
Building: 1 Floor: NA | Office: MOBILE  
Tel: 44-7967-275819  
e-mail: robin\_lyon@uk.ibm.com

#### Generelle Informationen:

**Das Gerät erfüllt die Schutzanforderungen nach EN 55024 und EN 55022 Klasse A.**

### Japan Voluntary Control Council for Interference Class A statement

**Attention:** This is a Class A product based on the standard of the Voluntary Control Council for Interference (VCCI). If this equipment is used in a domestic environment, radio interference may occur, in which case, the user may be required to take corrective actions.

この装置は、クラスA情報技術装置です。この装置を家庭環境で使用すると電波妨害を引き起こすことがあります。この場合には使用者が適切な対策を講ずるよう要求されることがあります。 VCCI-A

### Japan Electronics and Information Technology Industries Association (JEITA) statement

#### 高調波ガイドライン準用品

Japan Electronics and Information Technology Industries Association (JEITA) Confirmed Harmonics Guidelines with Modifications (products greater than 20 A per phase)

### Korean communications statement

Please note that this device has been approved for business purposes with regard to electromagnetic interference (Type A). If you find this is not suitable for your use, you may exchange it for a non-business purpose one.

이 기기는 업무용(A급)으로 전자파적합기기로서 판매자 또는 사용자는 이 점을 주의하시기 바라며, 가정외의 지역에서 사용하는 것을 목적으로 합니다.

### Russian Electromagnetic Interference (EMI) Class A statement

**ВНИМАНИЕ!** Настоящее изделие относится к классу А. В жилых помещениях оно может создавать радиопомехи, снижения которых необходимы дополнительные меры



## People's Republic of China Class A electronic emission Statement

**Attention:** This is a Class A product. In a domestic environment this product may cause radio interference, in which case the user may be required to take adequate measures.

### 中华人民共和国“A类”警告声明

#### 声 明

此为 A 级产品，在生活环境中，该产品可能会造成无线电干扰。在这种情况下，可能需要用户对其干扰采取切实可行的措施。

## Taiwan Class A compliance statement

#### 警告使用者：

這是甲類的資訊產品，在居住的環境中使用時，可能會造成射頻干擾，在這種情況下，使用者會被要求採取某些適當的對策。

## European Community (EC) Mark of Conformity Statement

This product is in conformity with the protection requirements of EC Council Directive 89/336/EEC on the approximation of the laws of the Member States relating to electromagnetic compatibility. Toshiba Global Commerce Solutions cannot accept responsibility for any failure to satisfy the protection requirements resulting from a non-recommended modification of the product, including the fitting of non-Toshiba Global Commerce Solutions option cards.

This product has been tested and found to comply with the limits for Class A Information Technology Equipment according to CISPR 22 / European Standard EN 55022. The limits for Class A equipment were derived for commercial and industrial environments to provide reasonable protection against interference with licensed communication equipment.

**Warning:** This is a Class A product. In a domestic environment this product may cause radio interference, in which case the user may be required to take adequate measures.

## Electrostatic Discharge (ESD)

**Attention:** Electrostatic discharge (ESD) damage can occur when there is a difference in charge between the part, the product, and the service person. No damage will occur if the service person and the part being installed are at the same charge level.

### ESD damage prevention

Anytime a service action involves physical contact with logic cards, modules, back-panel pins, or other ESD sensitive (ESDS) parts, the service person must be connected to an ESD common ground point on the product through the ESD wrist strap and cord.

The ESD ground clip can be attached to any frame ground, ground braid, green wire ground, or the round ground prong on the AC power plug. Coax or connector outside shells can also be used.

### Handling removed cards

Logic cards removed from a product should be placed in ESD protective containers. No other object should be allowed inside the ESD container with the logic card. Attach tags or reports that must accompany the card to the outside of the container.

## Japanese Electrical Appliance and Material Safety Law statement

本製品およびオプションに電源コードセットが付属する場合は、それぞれその装置専用のものになっていますので他の機器には使用しないで下さい。

## Japanese power line harmonics compliance statement

高調波ガイドライン適合品

高調波ガイドライン適合品

---

## Cable ferrite requirement

All cable ferrites are required to suppress radiated EMI emissions and must not be removed.

---

## Product recycling and disposal

This unit must be recycled or discarded according to applicable local and national regulations. Toshiba Global Commerce Solutions encourages owners of information technology (IT) equipment to responsibly recycle their equipment when it is no longer needed. Toshiba Global Commerce Solutions offers a variety of product return programs and services in several countries to assist equipment owners in recycling their IT products. Information on Toshiba Global Commerce Solutions product recycling offerings can be found on the Toshiba Global Commerce Solutions product recycling web site.

Español:

Esta unidad debe reciclarse o desecharse de acuerdo con lo establecido en la normativa nacional o local aplicable. Toshiba Global Commerce Solutions recomienda a los propietarios de equipos de tecnología de la información (TI) que reciclen responsablemente sus equipos cuando éstos ya no les sean útiles. Toshiba Global Commerce Solutions dispone de una serie de programas y servicios de devolución de productos en varios países, a fin de ayudar a los propietarios de equipos a reciclar sus productos de TI. Se puede encontrar información sobre las ofertas de reciclado de productos de Toshiba Global Commerce Solutions en el sitio web Toshiba Global Commerce Solutions product recycling.



**Notice:** This mark applies only to countries within the European Union (EU) and Norway.

Appliances are labeled in accordance with European Directive 2002/96/EC concerning waste electrical and electronic equipment (WEEE). The Directive determines the framework for the return and recycling of used appliances as applicable throughout the European Union. This label is applied to various products to indicate that the product is not to be thrown away, but rather reclaimed upon end of life per this Directive.

Remarque : Cette marque s'applique uniquement aux pays de l'Union Européenne et à la Norvège. L'étiquette du système respecte la Directive européenne 2002/96/EC en matière de Déchets des Equipements Electriques et Electroniques (DEEE), qui détermine les dispositions de retour et de recyclage applicables aux systèmes utilisés à travers l'Union européenne. Conformément à la directive, ladite étiquette précise que le produit sur lequel elle est apposée ne doit pas être jeté mais être récupéré en fin de vie.

注意：このマークは EU 諸国およびノルウェーにおいてのみ適用されます。

この機器には、EU 諸国に対する廃電気電子機器指令 2002/96/EC(WEEE) のラベルが貼られています。この指令は、EU 諸国に適用する使用済み機器の回収とリサイクルの骨子を定めています。このラベルは、使用済みになった時に指令に従って適正な処理をする必要があることを知らせるために種々の製品に貼られています。

In accordance with the European WEEE Directive, electrical and electronic equipment (EEE) is to be collected separately and to be reused, recycled, or recovered at end of life. Users of EEE with the WEEE marking per Annex IV of the WEEE Directive, as shown above, must not dispose of end of life EEE as unsorted municipal waste, but use the collection framework available to customers for the return, recycling, and recovery of WEEE. Customer participation is important to minimize any potential effects of EEE on the environment and human health due to the potential presence of hazardous substances in EEE. For proper collection and treatment, contact your local Toshiba Global Commerce Solutions representative.

Disposal of IT products should be in accordance with local ordinances and regulations.

---

## Battery return program

This product may contain sealed lead acid, nickel cadmium, nickel metal hydride, lithium, or lithium ion battery. Consult your user manual or service manual for specific battery information. The battery must be recycled or disposed of properly. Recycling facilities may not be available in your area. For information on disposal of batteries go to the Battery disposal web site or contact your local waste disposal facility.

### For Taiwan:



Please recycle batteries.

### For the European Union:



**Notice:** This mark applies only to countries within the European Union (EU)

Batteries or packaging for batteries are labeled in accordance with European Directive 2006/66/EC concerning batteries and accumulators and waste batteries and accumulators. The Directive determines the framework for the return and recycling of used batteries and accumulators as applicable throughout the European Union. This label is applied to various batteries to indicate that the battery is not to be thrown away, but rather reclaimed upon end of life per this Directive.

Les batteries ou emballages pour batteries sont étiquetés conformément aux directives européennes 2006/66/EC, norme relative aux batteries et accumulateurs en usage et aux batteries et accumulateurs usés. Les directives déterminent la marche à suivre en vigueur dans l'Union Européenne pour le retour et le recyclage des batteries et accumulateurs usés. Cette étiquette est appliquée sur diverses batteries pour indiquer que la batterie ne doit pas être mise au rebut mais plutôt récupérée en fin de cycle de vie selon cette norme.

バッテリーあるいはバッテリー用のパッケージには、EU 諸国に対する廃電気電子機器指令 2006/66/EC のラベルが貼られています。この指令は、バッテリーと蓄電池、および廃棄バッテリーと蓄電池に関するものです。この指令は、使用済みバッテリーと蓄電池の回収とリサイクルの骨子を定めているもので、EU 諸国にわたって適用されます。このラベルは、使用済みになったときに指令に従って適正な処理をする必要があることを知らせるために種々のバッテリーに貼られています。

In accordance with the European Directive 2006/66/EC, batteries and accumulators are labeled to indicate that they are to be collected separately and recycled at end of life. The label on the battery may also include a chemical symbol for the metal concerned in the battery (Pb for lead, Hg for mercury and Cd for

cadmium). Users of batteries and accumulators must not dispose of batteries and accumulators as unsorted municipal waste, but use the collection framework available to customers for the return, recycling and treatment of batteries and accumulators. Customer participation is important to minimize any potential effects of batteries and accumulators on the environment and human health due to the potential presence of hazardous substances. For proper collection and treatment, contact your local Toshiba Global Commerce Solutions representative.

This notice is provided in accordance with Royal Decree 106/2008 of Spain: The retail price of batteries, accumulators and power cells includes the cost of the environmental management of their waste.

## **For California:**

### **Perchlorate material – special handling may apply**

Refer to California Department of Toxic Substances Control.

The foregoing notice is provided in accordance with *California Code of Regulations Title 22, Division 4.5, Chapter 33: Best Management Practices for Perchlorate Materials*. This product/part includes a lithium manganese dioxide battery which contains a perchlorate substance.

---

## **Flat panel displays**

The fluorescent lamp in the liquid crystal display contains mercury. Dispose of it as required by local ordinances and regulations.

---

## **Monitors and workstations**

Connecticut: Visit the website of the Department of Energy & Environmental Protection for information about recycling covered electronic devices in the State of Connecticut, or telephone the Connecticut Department of Environmental Protection at 1-860-424-3000.

Oregon: For information regarding recycling covered electronic devices in the state of Oregon, go to the Oregon Department of Environmental Quality site.

Washington: For information about recycling covered electronic devices in the State of Washington, go to the Department of Ecology Web site or telephone the Washington Department of Ecology at 1-800-Recycle.

---

## Trademarks

The following are trademarks or registered trademarks of Toshiba, Inc. in the United States or other countries, or both:

Toshiba  
The Toshiba logo

The following are trademarks of Toshiba Global Commerce Solutions in the United States or other countries, or both:

AnyPlace  
SureMark  
SurePoint  
SurePOS  
TCxWave

The following are trademarks of International Business Machines Corporation in the United States or other countries, or both:

DB2  
DB2 Universal Database  
IBM and the IBM logo  
PS/2  
Wake on LAN  
WebSphere

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Magellan is a registered trademark of Datalogic Scanning, Inc.

SYMBOL a registered trademark of Symbol Technologies, Inc.

Microsoft, Windows, Windows NT, and the Windows 95 logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Celeron and Intel are trademarks of Intel corporation in the United States, or other countries.

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, or other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

---

# Glossary

This glossary includes terms and definitions from the *IBM Dictionary of Computing* (New York; McGraw-Hill, Inc., 1994).

## A

**ABM.** Asynchronous balanced mode.

**access method.** A software component in a processor for controlling the flow of information through a network.

**ACF/VTAM.** Advanced Communications Function for the Virtual Telecommunications Access Method.

**active.** (1) Able to communicate on the network. A token-ring network adapter is active if it is able to transmit and receive on the network. (2) Operational. (3) Pertaining to a node or device that is connected or is available for connection to another node or device. (4) Currently transmitting or receiving.

**adapter.** (1) In the point-of-sale terminal, a circuit card that, with its associated software, enables the terminal to use a function or feature. (2) In a LAN, within a communicating device, a circuit card that, with its associated software and/or microcode, enables the device to communicate over the network.

**adapter address.** Twelve hexadecimal digits that identify a LAN adapter.

**ADCS.** Advanced Data Communications for Stores

**address.** (1) In data communication, the IEEE-assigned unique code or the unique locally administered code assigned to each device or workstation connected to a network. (2) A character, group of characters, or a value that identifies a register, a particular part of storage, a data source, or a data link. The value is represented by one or more characters. (3) To refer to a device or an item of data by its address. (4) The location in the storage of a computer where data is stored.

**Advanced Data Communications for Stores (ADCS).** An IBM-licensed product that functions at the host processor to permit host-to-store communication.

**alert.** (1) An error message sent to the system services control point (SSCP) at the host system. (2) For IBM LAN management products, a notification indicating a possible security violation, a persistent error condition, or an interruption or potential interruption in the flow of data around the network. See also *network management vector transport*. (3) In SNA, a record sent to a system problem management focal point to communicate the existence of an alert condition. (4) In the NetView program, a high-priority event that warrants

immediate attention. This data base record is generated for certain event types that are designed by user-constructed filters.

**alphanumeric.** Pertaining to a character set containing letters, digits, and other special characters.

**Alphanumeric point-of-sale keyboard (ANPOS keyboard).** This keyboard consists of a section of alphanumeric keys, a programmable set of point-of-sale keys, a numeric keypad, and system function keys. If attached through the PS/2 port, this keyboard can optionally include a pointing device.

**alternate adapter.** In a personal computer that is used on a LAN and that supports installation of two network adapters, the adapter that uses alternate (not standard or default) mapping between adapter-shared RAM, adapter ROM, and designated computer memory segments. The alternate adapter is usually designated as adapter 1 in configuration parameters. Contrast with *primary adapter*.

**Alternate File Server.** A store controller that maintains image versions of all non-system mirrored files and that can assume control if the configured File Server becomes disabled.

**Alternate Master Store Controller.** The store controller that can take control of the LAN if the configured Master Store Controller becomes disabled. It maintains image versions of both system mirrored and system compound files.

**American National Standard Code for Information Interchange (ASCII).** The standard code, using a coded character set consisting of 7-bit coded characters (8 bits including parity check), used for information interchange among data processing systems, data communication systems, and associated equipment. The ASCII set consists of control characters and graphics characters.

**ANPOS keyboard.** Alphanumeric point-of-sale keyboard.

**API.** Application program interface.

**application program.** (1) A program written for or by a user that applies to the user's own work. (2) A program written for or by a user that applies to a particular application. (3) A program used to connect and communicate with stations in a network, enabling users to perform application-oriented activities.

**application program interface (API).** The formally defined programming language interface that is between an IBM system control program or a licensed program and the user of the program.



**architecture.** A logical structure that encompasses operating principles including services, functions, and protocols. See *computer architecture*, *network architecture*, *Systems Application Architecture (SAA)*, *Systems Network Architecture (SNA)*.

**ARTIC adapter.** A family of communications coprocessor adapters that, with appropriate electrical interfaces, can support a wide range of communication devices. For the Toshiba Store System, an ARTIC adapter provides communications support for ASYNC, SDLC, and X.25 communications.

**ASCII.** American National Standard Code for Information Interchange.

**async.** asynchronous.

**asynchronous (async).** (1) Pertaining to two or more processes that do not depend upon the occurrence of specific events such as timing signals. (2) Without regular time relationship; unexpected or unpredictable with respect to the execution of program instructions.

**asynchronous balanced mode (ABM).** An operational mode of a balanced data link in which either combined station can send commands at any time and can initiate transmission of response frames without explicit permission from the other combined station.

**attach.** (1) To connect a device physically. (2) To make a device a part of a network logically. Compare with *connect*.

**attaching device.** Any device that is physically connected to a network and can communicate over the network.

## B

**background.** On a color display, the part of the display screen that surrounds a character.

**background application.** A non-interactive program that can be selected from the background application screen or that can start automatically when the system is IPLed or when the controller is activated as the master or file server. Contrast with *foreground application*.

**backup.** Pertaining to a system, device, file, or facility that can be used in the event of a malfunction or the loss of data.

**bar code.** A code representing characters by sets of parallel bars of varying thickness and separation that are read optically by transverse scanning.

**baseband.** (1) A frequency band that uses the complete bandwidth of a transmission medium. Contrast with *broadband*, *carrierband*. (2) A method of data transmission that encodes, modulates, and impresses

information on the transmission medium without shifting or altering the frequency of the information signal.

**base unit.** The part of the 4683 Point-of-Sale terminal that contains the power supply and the interfaces.

**BASIC.** Beginner's All-purpose Symbolic Instruction Code. A programming language that uses common English words.

**basic conversation.** A conversation in which programs exchange data records in an SNA-defined format. This format is a stream of data containing 2-byte length prefixes that specify the amount of data to follow before the next prefix.

**batch.** Smaller subdivisions of price change records within an event. Each batch has a 12-character ID and a 30-character description field.

**baud.** The rate at which signal conditions are transmitted per second. Contrast with *bits per second (bps)*.

**beacon.** (1) A frame sent by an adapter on a ring network indicating a serious ring problem, such as a broken cable. It contains the addresses of the beaconing station and its nearest active upstream neighbor (NAUN). (2) To send beacon frames continuously. An adapter is *beaconing* if it is sending such a frame.

**beaconing.** An error-indicating function of token-ring adapters that assists in locating a problem causing a hard error on a token-ring network.

**binary.** (1) Pertaining to a system of numbers to the base two; the binary digits are 0 and 1. (2) Pertaining to a selection, choice, or condition that has two possible different values or states.

**bind.** In SNA products, a request to activate a session between two logical units.

**BIND.** See bind session.

**bind session (BIND).** In SNA products, a request to activate a session between two logical units (LUs).

**bit.** Either of the binary digits: a 0 or 1.

**bits per second (bps).** The rate at which bits are transmitted per second. Contrast with *baud*.

**block size.** (1) The minimum size that frames are grouped into for retransmission. (2) The number of data elements (such as bits, bytes, characters, or records) that are recorded or transmitted as a unit.

**bootstrap.** A sequence of instructions whose execution causes additional instructions to be loaded and executed until the complete computer program is in storage.



**bps.** Bits per second.

**Bps.** Bytes per second.

**bridge.** (1) An attaching device connected to two LAN segments to allow the transfer of information from one LAN segment to the other. A bridge may connect the LAN segments directly by network adapters and software in a single device, or may connect network adapters in two separate devices through software and use of a telecommunications link between the two adapters. (2) A functional unit that connects two LANs that use the same logical link control (LLC) procedures but may use the same or different medium access control (MAC) procedures. Contrast with *gateway* and *router*.

**broadband.** A frequency band divisible into several narrower bands so that different kinds of transmissions such as voice, video, and data transmission can occur at the same time. Synonymous with *wideband*. Contrast with *baseband*.

**buffer.** (1) A portion of storage used to hold input or output data temporarily. (2) A routine or storage used to compensate for a difference in data rate or time of occurrence of events, when transferring data from one device to another.

**bus.** (1) In a processor, a physical facility on which data is transferred to all destinations, but from which only addressed destinations may read in accordance with appropriate conventions. (2) A network configuration in which nodes are interconnected through a bidirectional transmission medium. (3) One or more conductors used for transmitting signals or power.

**byte.** A string consisting of 8 bits that is treated as a unit, and that represents a character. See *n-bit byte*.

## C

**C.** A high-level programming language designed to optimize run time, size, and efficiency.

**C & SM.** Communications and systems management.

**cable loss (optical).** The loss in an optical cable equals the attenuation coefficient for the cables fiber times the cable length.

**cable segment.** A section of cable between components or devices on a network. A segment may consist of a single patch cable, multiple patch cables connected together, or a combination of building cable and patch cables connected together. See *LAN segment*, *ring segment*.

**call.** The action of bringing a function or subprogram into effect, usually by specifying the entry conditions and jumping to an entry point.

**carrierband.** A frequency band in which the modulated signal is superimposed on a carrier signal (as differentiated from baseband), but only one channel is present on the medium. Contrast with *baseband*, *broadband*.

**cash drawer.** A drawer at a point-of-sale terminal that can be programmed to open automatically. See *till*.

**CCB.** Command control block.

**CCC/IP.** Controller-to-Controller Communications over Internet Protocol.

**CCITT.** Comité Consultatif International Télégraphique et Téléphonique. The International Telegraph and Telephone Consultative Committee.

**CD.** Corrective diskette.

**CD-ROM.** Compact disc Read-only memory. High-capacity read-only memory in the form of an optically read compact disk.

**chain.** (1) Transfer of control from the currently executing program to another program or overlay. (2) Referencing a data record from a previous data record.

**channel.** (1) A functional unit, controlled by a host computer, that handles the transfer of data between processor storage and local peripheral equipment. (2) A path along which signals can be sent. (3) The portion of a storage medium that is accessible to a given reading or writing station.

**CICS.** Customer Information Control System.

**circuit.** (1) A logic device. (2) One or more conductors through which an electric current can flow.

**class.** (1) A template for creating objects; a class defines data and methods; a class is a unit of organization in a Java program. A class can pass on its public data and methods to its subclasses. (2) A collection of variables and methods that an object can have, or a template for building objects.

**.class file.** A file containing machine-independent Java bytecodes. The Java compiler generates *.class* files for the Java interpreter to read.

**class method.** A class method is a function that is defined as a part of a class.

**classpath.** An environment variable used to define all the directories where *.class* files are found.

**.class variable.** A variable allocated once per class. Class variables have global class scope and belong to the entire class instead of an instance.

**clear.** To delete data from a screen or from memory.

**COBOL.** Common business-oriented language. A high-level programming language, based on English, that is used primarily for business applications.

**command.** (1) A request for performance of an operation or execution of a program. (2) A character string from a source external to a system that represents a request for system action.

**command control block (CCB).** In the Token-Ring Network, a specifically formatted information provided from the application program to the adapter support software to request an operation.

**Common Programming Interface-Communications (CPI-C).** Provides languages, commands, and calls that allow the development of applications that are more easily integrated and moved across environments supported by Systems Applications Architecture (SAA).

**communication adapter.** A circuit card and its associated software that enable a device, such as a personal computer, to be connected to a network or another computer (examples include binary synchronous, asynchronous, modem, and LAN adapters).

**communications and systems management (C & SM).** A set of tools, programs, and network functions used to plan, operate, and control an SNA communications network. C & SM runs on the store controller and must also exist at the host site.

**compact disc- read-only memory (CD-ROM).** (1) A 4.75-inch optical memory storage medium, capable of storing approximately 650 megabytes of data. Data is read optically by means of a laser. (2) A disc with information stored in the form of pits along a spiral track. The information is decoded by a compact-disc player and interpreted as digital audio data, which most computers can process.

**compile.** (1) To translate all or part of a program expressed in a high-level language into a computer program expressed in an intermediate language, an assembly language, or a machine language. (2) To prepare a machine language program from a computer program written in another programming language by making use of the overall logic structure of the program, or generating more than one computer instruction for each symbolic statement, or both, as well as performing the function of an assembler. (3) To translate a source program into an executable program (an object program). (4) To translate a program written in a high-level programming language into a machine language program.

**compound files.** Files that are kept on all store controllers.

**computer architecture.** The organizational structure of a computer system, including hardware and software.

**concurrent conversations.** The ability of a transaction program (TP) to manage more than one LU 6.2 conversation at the same time. When this ability is written into a TP, the TP is said to be *managing concurrent conversations*.

**configuration.** The group of devices, options, and programs that make up a data processing system or network as defined by the nature, number, and chief characteristics of its functional units. More specifically, the term may refer to a hardware configuration or a software configuration. See also *system configuration*.

**configuration parameters.** Variables in a configuration definition, the values of which characterize the relationship of a product, such as a bridge, to other products in the same network.

**connect.** In a LAN, to physically join a cable from a station to an access unit or network connection point. Contrast with *attach*.

**contention.** In a LAN, a situation in which two or more data stations are allowed by the protocol to start transmitting concurrently and thus risk collision.

**contention loser.** In APPC, the LU that must request and receive permission from the session partner LU to allocate a session. Contrast with *contention winner*.

**contention winner.** The LU that can allocate a session without requesting permission from the session partner LU. Contrast with *contention loser*.

**contiguous.** Touching or joining at the edge or boundary; adjacent. For example, an unbroken consecutive series of memory locations.

**controller.** A unit that controls input/output operations for one or more devices.

**conversation.** A logical connection between two programs over an LU type 6.2 session that allows them to communicate with each other while processing a transaction. See also *basic conversation* and *mapped conversation*.

**conversation partner.** One of the two programs involved in a conversation.

**conversation state.** The condition of a conversation that reflects what the past action on that conversation has been and that determines what the next set of actions may be.

**corrective diskette (CD).** A set of diskettes that contain modules to replace the modules in the active program subdirectory. The first diskette of the set must contain a product control file that describes which product the modules are to be applied to and a list of all modules that are to be replaced.

**CRC.** Cyclic redundancy check.

**cursor.** A movable point of light (or a short line) that indicates where the next character is to be entered on the display screen.

**Customer Information Control System (CICS).** An IBM licensed program that enables transactions entered at remote terminals to be processed concurrently by user-written application programs. It includes facilities for building, using, and maintaining data bases.

**customer receipt.** An itemized list of merchandise purchased and paid for by the customer.

**customize.** To tailor a program or store system through option selection.

**cyclic redundancy check (CRC).** Synonym for *frame check sequence (FCS)*.

## D

**data.** (1) A representation of facts, concepts, or instructions in a formalized manner suitable for communication, interpretation, or processing by human or automatic means. (2) Any representations such as characters or analog quantities to which meaning is or might be assigned.

**data circuit-terminating equipment (DCE).** In a data station, the equipment that provides the signal conversion and coding between the data terminal equipment (DTE) and the line.

**data communication.** (1) Transfer of information between functional units by means of data transmission according to a protocol. (2) The transmission, reception, and validation of data.

**data file.** A collection of related data records organized in a specific manner; for example, a payroll file (one record for each employee, showing such information as rate of pay and deductions) or an inventory file (one record for each inventory item, showing such information as cost, selling price, and number in stock.) See also *data set*, *file*.

**data link.** (1) Any physical link, such as a wire or a telephone circuit, that connects one or more remote terminals to a communication control unit, or connects one communication control unit with another. (2) The assembly of parts of two data terminal equipment (DTE) devices that are controlled by a link protocol, and the interconnecting data circuit, that enable data to be transferred from a data source to a data link. (3) In SNA, see also *link*. **Note:** A telecommunication line is only the physical medium of transmission. A data link includes the physical medium of transmission, the protocol, and associated devices and programs; it is both physical and logical.

**data processing system.** A network, including computer systems and associated personnel, that

accepts information, processes it according to a plan, and produces the appropriate results.

**data rate.** See *data transfer rate*, *line data rate*.

**data set.** Logically related records treated as a single unit. See also *file*.

**data terminal equipment (DTE).** (1) That part of a data station that serves as a data source, data receiver, or both. (2) Equipment that sends or receives data, or both.

**data transfer.** (1) The result of the transmission of data signals from any data source to a data receiver. (2) The movement, or copying, of data from one location and the storage of the data at another location.

**data transfer rate.** The average number of bits, characters, or blocks per unit of time passing between equipment in a data-transmission session. The rate is expressed in bits, characters, or blocks per second, minute, or hour.

**data transmission.** The conveying of data from one place for reception elsewhere by means of telecommunications.

**dB.** Decibel.

**DBCS.** Double-byte character set.

**DCE.** Data circuit-terminating equipment.

**DDA.** Data Distribution Application.

**debug.** To detect, diagnose, and eliminate errors in computer programs.

**decibel (dB).** (1) One tenth of a bel. (2) A unit that expresses the ratio of two power levels on a logarithmic scale. (3) A unit for measuring relative power. The number of decibels is 10 times the logarithm base (base 10) of the ratio of the measured power levels; if the measured levels are voltages (across the same or equal resistance), the number of decibels is twenty times the log of the ratio. See also *neper*.

**default.** Pertaining to an attribute, value, or option that is assumed when none is explicitly specified.

**default value.** The value the system supplies when the user does not specify a value.

**delayed data maintenance.** A function that allows the item record, the operator and the check authorization files to be maintained from the host on an immediate or a delayed basis.

**destination.** Any point or location, such as a node, station, or particular terminal, to which information is to be sent.

**device.** (1) A mechanical, electrical, or electronic contrivance with a specific purpose. (2) An input/output unit such as a terminal, display, or printer. See also *attaching device*.

**device channel.** In Toshiba Point-of-Sale terminals, a path along which signals for serial input/output devices can be sent. For these terminals, the device channel controller or adapter is contained on the system board.

**diagnostic diskette.** A diskette containing diagnostic modules or tests used by computer users and service personnel to diagnose hardware problems.

**diagnostics.** Modules or tests used by computer users and service personnel to diagnose hardware problems.

**dialing.** Using a dial or pushbutton telephone to initiate a telephone call. In telecommunication, attempting to establish a connection between a terminal and a telecommunication device over a switched line.

**direct memory access (DMA).** A procedure or method designed to transfer data between main storage and I/O units without intervention of the processing unit.

**directory.** (1) A table of identifiers and references that correspond to items of data. (2) An index that a control program uses to locate one or more blocks of data that are stored in separate areas of a data set in direct access storage.

**disabled.** (1) Pertaining to a state of a processing unit that prevents the occurrence of certain types of interruptions. (2) Pertaining to the state in which a transmission control unit or audio response unit cannot accept incoming calls on a line.

**DISC.** Disconnect character.

**disk.** A round, flat plate coated with a magnetic substance on which computer data is stored. See also *integrated disk*, *fixed disk*.

**diskette.** A thin, flexible magnetic disk permanently enclosed in a protective jacket. A diskette is used to store information for processing.

**Disk Operating System (DOS).** An operating system for computer systems that use disks and diskettes for auxiliary storage of programs and data.

**display.** (1) A visual presentation of data. (2) A device that presents visual information to the point-of-sale terminal operator and to the customer, or to the display station operator.

**distributed.** Physically separate but connected by cables.

**Distributed Systems Executive (DSX).** An IBM licensed program available for IBM host systems that

allows the host system to get, send, and remove files, programs, formats and procedures in a network of computers.

**DMA.** Direct memory access

**domain.** An SSCP and the resources that it can control.

**DOS.** Disk Operating System.

**double-byte character set (DBCS).** A set of characters in which each character is represented by 2 bytes. Languages such as Japanese, Chinese, and Korean, which contain more symbols than can be represented by 256 code points, require double-byte character sets. Because each character requires 2 bytes, the typing, display, and printing of DBCS characters requires hardware and programs that support DBCS. Contrast with single-byte character set.

**driver.** Software component that controls a device.

**drop.** A cable that leads from a faceplate to the distribution panel in a wiring closet. When the IBM Cabling System is used with the Token-Ring Network, a drop may form part of a lobe. See also *lobe*.

**DSX.** Distributed Systems Executive.

**DTE.** Data terminal equipment.

**dump.** (1) To write at a particular instant the contents of storage, or part of storage, onto another data medium for the purpose of safeguarding or debugging the data. (2) Data that has been dumped.

## E

**EAN.** European article number.

**EBCDIC.** Extended binary-coded decimal interchange code.

**EIA.** Electronic Industries Association. See *EIA interface*.

**EIA interface.** An industry-accepted interface for connecting devices having voltage-related limits.

**emulation.** (1) The imitation of all or part of one computer system by another, primarily by hardware, so that the imitating system accepts the same data, executes the same programs, and achieves the same results as the imitated computer system. (2) The use of programming techniques and special machine features to permit a computing system to execute programs written for another system.

**enabled.** (1) On a LAN, pertaining to an adapter or device that is active, operational, and able to receive frames from the network. (2) Pertaining to a state of a processing unit that allows the occurrence of certain



types of interruptions. (3) Pertaining to the state in which a transmission control unit or an audio response unit can accept incoming calls on a line.

**envelope.** (1) Information added to a frame or other message unit to allow it to be transmitted using a protocol other than the protocol in which the message unit originated. (2) To surround or enclose a message unit in information to allow the message unit to be transmitted using a protocol other than the protocol in which the message originated.

**error condition.** The condition that results from an attempt to use instructions or data that are invalid.

**error message.** A message that is issued because an error has been detected.

**Ethernet.** A 10-megabit baseband local area network that allows multiple stations to access the transmission medium at will without prior coordination, avoids contention by using carrier sense and deference, and resolves contention by using collision detection and transmission. Ethernet uses carrier sense multiple access with collision detection (CSMA/CD).

**European article number (EAN).** A number that is assigned to and encoded on an article of merchandise for scanning in some countries.

**evaluation.** Reduction of an expression to a single value.

**exchange identification (XID).** The ID that is exchanged with the remote physical unit when an attachment is first established.

**execute.** To perform the actions specified by a program or a portion of a program.

**execution.** The process of carrying out an instruction or instructions of a computer program by a computer.

**exit.** To execute an instruction or statement within a portion of a program in order to terminate the execution of that portion. **Note:** Such portions of programs include loops, routines, subroutines, and modules.

**expansion board.** In a personal computer, a panel containing microchips that a user can install in an expansion slot to add memory or special features. Synonymous with *expansion card*, *extender card*.

**expansion card.** Synonym for *expansion board*.

**expansion slot.** In a personal computer, one of several receptacles in the system board of the system unit or expansion unit into which a user can install an expansion board such as a memory expansion option.

**extended binary-coded decimal interchange code (EBCDIC).** A coded character set consisting of 8-bit coded characters.

**extender card.** Synonym for *expansion board*.

## F

**fault.** An accidental condition that causes a functional unit to fail to perform its required function.

**feature.** A part of a Toshiba product that may be ordered separately by the customer.

**Feature Expansion.** A card that plugs into an 4683 Point-of-Sale Terminal and allows additional devices to be used.

**field.** On a data medium or a storage medium, a specified area used for a particular category of data; for example, a group of character positions used to enter or display wage rates on a panel.

**file.** A named set of records stored or processed as a unit. For example, an invoice may form a record and the complete set of such records may form a file. See also *data file* and *data set*.

**file name.** (1) A name assigned or declared for a file. (2) The name used by a program to identify a file.

**file server.** (1) A store controller that maintains prime versions of all non-system mirrored files. (2) A high-capacity disk storage device or a computer that each computer on a network can access to retrieve files that can be shared among the attached computers.

**file type.** The attribute of a file that specifies to which store controllers it is distributed.

**fixed disk (drive).** In a personal computer system unit, a disk storage device that reads and writes on rigid magnetic disks. It is faster and has a larger storage capacity than a diskette and is permanently installed.

**foreground.** On a color display, the part of the display area that is the character itself.

**foreground application.** An interactive program that can be selected by system menus or started in command mode. Contrast with *background application*.

**formatted diskette.** A diskette on which track and sector control information has been written and that can be used by the computer to store data. **Note:** A diskette must be formatted before it can receive data.

**frame.** (1) The unit of transmission in some LANs, including the Token-Ring Network. It includes delimiters, control characters, information, and checking characters. On a token-ring network, a frame is created from a token when the token has data appended to it. On a token-bus network, all frames including the token frame contain a preamble, start delimiter, control address, optional data and checking characters, end delimiter, and are followed by a minimum silence period. (2) A housing for machine elements. (3) In synchronous

data link control (SDLC), the vehicle for every command, every response, and all information that is transmitted using SDLC procedures. Each frame begins and ends with a flag.

**frame check sequence (FCS).** (1) A system of error checking performed at both the sending and receiving station after a block-check character has been accumulated. (2) A numeric value derived from the bits in a message that is used to check for any bit errors in transmission. (3) A redundancy check in which the check key is generated by a cyclic algorithm. Synonymous with *cyclic redundancy check (CRC)*.

**franking.** Printing an indication on a document that the document has been processed. This franking may be a store header line, a "total" line, or a transaction number that is printed when a check, a discount coupon, or a gift certificate is inserted in the document insert station of the point-of-sale terminal during certain types of transactions.

**frequency.** The rate of signal oscillation, expressed in hertz (cycles per second).

**function.** (1) A specific purpose of an entity, or its characteristic action. (2) A subroutine that returns the value of a single variable. (3) In data communications, a machine action such as a carriage return or line feed.

## G

**gateway.** A device and its associated software that interconnect networks of systems of different architectures. The connection is usually made above the Reference Model network layer. For example, a gateway allows LANs access to System/370 host computers. Contrast with *bridge* and *router*.

**group.** (1) A set of related records that have the same value for a particular field in all records. (2) A collection of users who can share access authorities for protected resources. (3) A list of names that are known together by a single name.

## H

**hardware.** Physical equipment as opposed to programs, procedures, rules, and associated documentation.

**HCP.** Host command processor for advanced data communications.

**HCP.** Host command processor.

**header.** The portion of a message that contains control information for the message such as one or more destination fields, name of the originating station, input sequence number, character string indicating the type of message, and priority level for the message.

**host application program.** An application program that the host processor executes.

**host command processor (HCP).** The SNA logical unit of the programmable Store System store controller.

**host computer.** (1) The primary or controlling computer in a multi-computer installation or network. (2) In a network, a processing unit in which resides a network access method. Synonymous with *host processor*.

**host processor.** (1) In a network, a computer that primarily provides services such as computation, data base access, or special programs or programming languages. (2) Synonym for *host computer*.

**host processor.** (1) A processor that controls all or part of a user application network. (2) In a network, the processing unit in which resides the access method for the network. (3) In an SNA network, the processing unit that contains a system services control point (SSCP). (4) A processing unit that executes the access method for attached communication controllers. (5) The processing unit required to create and maintain PSS. Synonymous with *host computer*.

## I

**IBM Disk Operating System (DOS).** A disk operating system based on MS-DOS\*\*.

**identifier.** String of characters used to name elements of a program, such as variable names, reserved words, and user-defined function names.

**idles.** Signals sent along a ring network when neither frames nor tokens are being transmitted.

**image version.** Copy of a prime version of a file. See *prime version*.

**inactive.** (1) Not operational. (2) Pertaining to a node or device not connected or not available for connection to another node or device. (3) In the Token-Ring Network, pertaining to a station that is only repeating frames or tokens, or both.

**information (I) frame.** A frame in I format used for numbered information transfer. See also *supervisory frame*, *unnumbered frame*.

**initialize.** In a LAN, to prepare the adapter (and adapter support code, if used) for use by an application program.

**initial program load (IPL).** The initialization procedure that causes an operating system to begin operation.

**input device.** Synonym for *input unit*.

**input field.** An unprotected display field that the terminal operator can add to, modify, or erase by using the keyboard. Contrast with *protected field*.

**input/output (I/O).** (1) Pertaining to a device whose parts can perform an input process and an output process at the same time. (2) Pertaining to a functional unit or channel involved in an input process, output process, or both, concurrently or not, and to the data involved in such a process.

**input unit.** A device in a data processing system by means of which data can be entered into the system. Synonymous with *input device*.

**insert.** To make an attaching device an active part of a LAN.

**integrated.** Arranged together as one unit.

**integrated disk.** An integral part of the processor that is used for magnetically storing files, application programs, and diagnostics. Synonymous with *disk*.

**interactive.** Pertaining to an application or program in which each entry calls forth a response from a system or program. An interactive program may also be conversational, implying a continuous dialog between the user and the system.

**interface.** (1) A shared boundary between two functional units, defined by functional characteristics, common physical interconnection characteristics, signal characteristics, and other characteristics as appropriate. (2) A shared boundary. An interface may be a hardware component to link two devices or a portion of storage or registers accessed by two or more computer programs. (3) Hardware, software, or both, that links systems, programs, or devices.

**interference.** (1) The prevention of clear reception of broadcast signals. (2) The distorted portion of a received signal.

**interleave.** To insert segments of one program into another program so that the two programs can, in effect, be executed at the same time.

**interrupt.** (1) A suspension of a process, such as execution of a computer program, caused by an external event and performed in such a way that the process can be resumed. (2) To stop a process in such a way that it can be resumed. (3) In data communication, to take an action at a receiving station that causes the sending station to end a transmission. (4) A means of passing processing control from one software or microcode module or routine to another, or of requesting a particular software, microcode, or hardware function.

**interrupt level.** The means of identifying the source of an interrupt, the function requested by an interrupt, or the code or feature that provides a function or service.

**I/O.** Input/output.

**I/O device.** Equipment for entering and receiving data from the system.

**IP.** Internet Protocol.

**IPL.** Initial program load.

**isochronous.** Time-dependent. Refers to processes in which data must be delivered within certain time constraints.

**item.** (1) One member of a group. (2) In a store, one unit of a commodity, such as one box, one bag, or one can. Usually an item is the smallest unit of a commodity to be sold.

## J

**Java.** An object-oriented programming language designed to be platform independent.

**Java application.** A Java Virtual Machine (JVM) combined with its class and parameters.

**Java Virtual Machine (JVM).** Java interpreter that runs the class.

**jumper.** A connector between two pins on a network adapter that enables or disables an adapter option, feature, or parameter value.

**JUCC.** Japan Unified Cash Card.

**JVM.** See Java Virtual Machine.

## K

**K.** When referring to storage capacity, a symbol that represents two to the tenth power, or 1024.

**Kb.** Kilobit.

**KB.** Kilobyte.

**keyboard.** A group of numeric keys, alphabetic keys, special character keys, or function keys used for entering information into the terminal and into the system.

**keyed file.** Type of file composed of keyed records. Each keyed record has two parts: a key and data. A key is used to identify and access each record in the file.

**kilobit (Kb).** 1024 binary digits.

**kilobyte (KB).** 1024 bytes for processor and data storage (memory) size.

## L

**label.** Constant, either numeric or literal, that references a statement or function.

**LAN.** Local area network.

**LAN segment.** (1) Any portion of a LAN (for example, a single bus or ring) that can operate independently but is connected to other parts of the establishment network by bridges. (2) An entire ring or bus network without bridges. See *cable segment*, *ring segment*.

**LCD.** Liquid crystal display.

**leased line.** Synonym for *nonswitched line*.

**LED.** Light-emitting diode.

**light-emitting diode (LED).** A semiconductor chip that gives off visible or infrared light when activated.

**line connection.** In the Toshiba Store System, the physical connection (or equipment) between nodes that provides two-way communication and error correction and detection between one link station and one or more other link stations. **Note:** In SNA, this physical connection is called a *link connection*. In the Toshiba Store System, it is called a *line connection*.

**line data rate.** The rate of data transmission over a telecommunications link.

**link.** (1) In the Toshiba Store System, the logical connection between nodes including the end-to-end link control procedures. (2) The combination of physical media, protocols, and programming that connects devices on a network. (3) In computer programming, the part of a program, in some cases a single instruction or an address, that passes control and parameters between separate portions of the computer program. (4) To interconnect items of data or portions of one or more computer programs. (5) In SNA, the combination of the link connection and link stations joining network nodes. See also *link connection*. **Note:** A link connection is the physical medium of transmission; for example, a telephone wire or a microwave beam. A link includes the physical medium of transmission, the protocol, and associated devices and programming; it is both logical and physical.

**link connection.** (1) All physical components and protocol machines that lie between the communicating link stations of a link. The link connection may include a switched or leased physical data circuit, a LAN, or an X.25 virtual circuit. (2) In SNA, the physical equipment providing two-way communication and error correction and detection between one link station and one or more other link stations. (3) In the Toshiba Store System, the logical link providing two-way communication of data from one network node to one or more other network nodes.

**listing.** A printout of source code.

**load.** In computer programming, to enter data into memory or working registers.

**lobe.** In the Token-Ring Network, the section of cable (which may consist of several segments) that connects an attaching device to an access unit.

**local area network (LAN).** A computer network located on a user's premises within a limited geographical area. **Note:** Communication within a LAN is not subject to external regulations; however, communication across the LAN boundary may be subject to some form of regulation.

**local program.** The program being discussed within a particular context. Contrast with *remote program*.

**logical file name (LFN).** An abbreviated file name used to represent either an entire file name or the drive and subdirectory path part of the file name.

**logical link.** In an MVS/VS multisystem environment, the means by which a physical link is related to the transactions and terminals that can use the physical link.

**logical unit (LU).** (1) In SNA, a port through which an end user accesses the SNA network in order to communicate with another end user and through which the end user accesses the functions provided by system services control points (SSCPs). An LU can support at least two sessions, one with an SSCP and one with another LU, and may be capable of supporting many sessions with other logical units. (2) A type of network addressable unit that enables end users to communicate with each other and gain access to network resources.

**logon (n).** The procedure for starting up a point-of-sale terminal or store controller for normal sales operations by sequentially entering the correct security number and transaction number. Synonymous with *sign-on*.

**log on (v).** (1) To initiate a session. (2) In SNA products, to initiate a session between an application program and a logical unit (LU). Synonymous with *sign-on*.

**loop.** (1) A set of instructions that may be executed repeatedly while a certain condition prevails. See also *store loop*. (2) A closed unidirectional signal path connecting input/output devices to a network.

**LU.** Logical unit.

## M

**magnetic stripe.** The magnetic material (similar to recording tape) on merchandise tickets, credit cards, and employee badges. Information is recorded on the



stripe for later “reading” by the magnetic stripe reader (MSR) or magnetic wand reader attached to the point-of-sale terminal.

**magnetic stripe reader (MSR).** A device that reads coded information from a magnetic stripe on a card, such as a credit card, as it passes through a slot in the reader.

**maintenance analysis procedure (MAP).** Deprecated term for *procedure*. See *procedure*.

**maintenance diskette.** See *corrective diskette*.

**Manufacturing Automated Protocol (MAP).** A broadband LAN with a bus topology that passes tokens from adapter to adapter on a coaxial cable.

**MAP.** (1) Maintenance analysis procedure. (2) Manufacturing Automated Protocol.

**mapped conversation.** A conversation in which programs exchange data records with arbitrary data formats agreed upon by the applications programmers.

**mapping.** Establishing a correspondence between the elements of one set and the elements of another set.

**master store controller.** The store controller that maintains prime versions of system mirrored files and all compound files.

**Mb.** Megabit.

**MB.** Megabyte.

**MCF Network.** Multiple store controllers communicating on a network using DDA. This provides data redundancy among the store controllers.

**media.** Plural form of *medium*.

**medialess.** Not fitted with a direct access storage device, such as a diskette drive or fixed disk drive, as in some models of Toshiba Point of Sale Terminals.

**medium.** (1) A physical carrier of electrical or optical energy. (2) A physical material in or on which data may be represented.

**megabit (Mb).** A unit of measure for throughput. 1 megabit = 1,048,576 bits.

**megabyte (MB).** A unit of measure for data. 1 megabyte = 1,048,576 bytes.

**megahertz (MHz).** A unit of measure of frequency. 1 megahertz = 1,000,000 hertz.

**memory.** Program-addressable storage from which instructions and other data can be loaded directly into registers for subsequent execution or processing.

**message.** (1) An arbitrary amount of information whose beginning and end are defined or implied. (2) A

group of characters and control bit sequences transferred as an entity. (3) In telecommunication, a combination of characters and symbols transmitted from one point to another. (4) A logical partition of the user device’s data stream to and from the adapter. See also *error message*, *operator message*.

**MHz.** Megahertz.

**Micro Channel.** The architecture used by IBM Personal System/2 computers, Models 50 and above. This term is used to distinguish these computers from personal computers using a PC I/O channel, such as an IBM PC, XT, or an IBM Personal System/2 computer, Model 25 or 30.

**migration.** Upgrade of a program to a newer version or release.

**mirrored files.** Files that are kept on both the Master Store Controller and the Alternate Master Store Controller or on both the File Server and Alternate File Server. System mirrored files are kept on the Master Store Controller and Alternate Store Controller and non-system mirrored files are kept on the File Server and Alternate File Server.

**Mod1.** A generic name used to refer to a point-of-sale terminal in the 4690 Store System that loads and executes programs. A Mod1 can be any of the following models: 4683-001, 4683-A01, 4683-P11, 4683-P21, 4683-P41, 4683-421, 4693-xx1, and 4694-xx4 (terminal part if a controller/terminal).

**Mod2.** A generic name used to refer to a point-of-sale terminal in the 4690 Store System that does not load and execute programs, but attaches to a terminal that does. A Mod2 can be one of the following models: 4683-002, 4683-A02, or 4693-2x2.

**modem (MOdulator/DEModulator).** A device that converts digital data from a computer to an analog signal that can be transmitted in a telecommunication line, and converts the analog signal received to data for the computer.

**module.** A program unit that is discrete and identifiable with respect to compiling, combining with other units, and load; for example, the input to, or output from, an assembler, compiler, linkage editor, or executive routine.

**modulo check.** A function designed to detect most common input errors by performing a calculation on values entered into a system by an operator or scanning device.

**monitor.** (1) A functional unit that observes and records selected activities for analysis within a data processing system. Possible uses are to show significant departures from the norm, or to determine levels of utilization of particular functional units. (2) Software or hardware that observes, supervises, controls, or verifies operations of a system.

**monochrome display.** A display device that presents display images in only one color.

**MSR.** Magnetic stripe reader.

**multiple controller system.** Synonym for *MCF Network*.

**multipoint.** Pertaining to communication among more than two stations over a single telecommunication line.

**multipoint line.** A telecommunication line or circuit connecting two or more stations. Contrast with *point-to-point line*.

## N

**name.** An alphanumeric term that identifies a data set, statement, program, or cataloged procedure.

**n-bit byte.** A string that consists of n bits.

**NCP.** Network control program.

**neper.** A unit for measuring power. The number of nepers is the logarithm (base e) of the ratio of the measured power level.

**NetBIOS.** Network Basic Input/Output System.

**NetView.** A host-based IBM network management licensed program that provides communication network management (CNM) or communications and systems management (C & SM) services.

**NetView Distribution Manager (NetView DM).** A component of the NetView family supporting resource distribution within *Change Management*, and providing central control of software and microcode distribution and installation, to processors in a distributed/departmental (SNA) network system. It allows a similar control of user data objects across the network, and provides the facilities to support the remote initiation of command lists.

**network.** (1) A configuration of data processing devices and software connected for information interchange. (2) An arrangement of nodes and connecting branches. Connections are made between data stations.

**network administrator.** A person who manages the use and maintenance of a network.

**network architecture.** The logical structure and operating principles of a computer network. See also *systems network architecture (SNA)* and *Open Systems Interconnect (OSI) architecture*. **Note:** The operating principles of a network include those of services, functions, and protocols.

**Network Basic Input/Output System (NetBIOS).** A message interface used on LANs to provide message,

print server, and file server functions. The IBM NetBIOS application program interface (API) provides a programming interface to the LAN so that an application program can have LAN communication without knowledge and responsibility of the data link control (DLC) interface.

**network control program (NCP).** A control program for the 3704 or 3705 Communications Controller, generated by the user from a library of Toshiba-supplied modules.

**network file system (NFS).** A system that allows you to mount remote file systems across homogeneous and heterogeneous systems.

**network management vector transport (NMVT).** The portion of an alert transport frame that contains the alert message.

**NFS.** network file system

**node.** (1) Any device, attached to a network, that transmits and/or receives data. (2) An end point of a link, or a junction common to two or more links in a network. Nodes can be processors, controllers, or workstations. Nodes can vary in routing and other functional capabilities. (3) In a network, a point where one or more functional units interconnect transmission lines.

**node address.** The address of an adapter on a LAN.

**nonswitched line.** (1) A connection between systems or devices that does not have to be made by dialing. Contrast with *switched line*. (2) A telecommunication line on which connection does not have to be established by dialing. Synonymous with *leased line*.

**nonvolatile random access memory (NVRAM).** Random access memory that retains its contents after electrical power is shut off.

**NRZI.** (1) Non-return-to-zero inverted transmission. (2) Non-return-to-reference transmission in which the zeros are represented by a bit cell boundary transition in the information signal, and ones are represented by the absence of a bit cell boundary transition.

**NVRAM.** nonvolatile random access memory

## O

**OCR.** Optical character recognition.

**offline.** Operation of a functional unit without the control of a computer or control unit.

**online.** Operation of a functional unit that is under the continual control of a computer or control unit. The term also describes a user's access to a computer using a terminal.

**open.** (1) To make an adapter ready for use. (2) A break in an electrical circuit. (3) To make a file ready for use.

**Open Systems Interconnect (OSI).** (1) The interconnection of open systems in accordance with specific ISO standards. (2) The use of standardized procedures to enable the interconnection of data processing systems. **Note:** OSI architecture establishes a framework for coordinating the development of current and future standards for the interconnection of computer systems. Network functions are divided into seven layers. Each layer represents a group of related data processing and communication functions that can be carried out in a standard way to support different applications.

**Open Systems Interconnect (OSI) architecture.** Network architecture that adheres to a particular set of ISO standards that relates to Open Systems Interconnect (OSI).

**Open Systems Interconnect (OSI) Reference Model.** A model that represents the hierarchical arrangement of the seven layers described by the Open Systems Interconnect (OSI) architecture.

**operating system.** Software that controls the execution of programs. An operating system may provide services such as resource allocation, scheduling, input/output control, and data management. Examples are IBM DOS and IBM OS/2.

**Operating System/2 (OS/2).** A set of programs that control the operation of high-speed large-memory IBM Personal Computers (such as the IBM Personal System/2 computer, Models 50 and above), providing multitasking and the ability to address up to 16 MB of memory. Contrast with *Disk Operating System (DOS)*.

**operation.** (1) A defined action, namely, the act of obtaining a result from one or more operands in accordance with a rule that completely specifies the result for any permissible combination of operands. (2) A program step undertaken or executed by a computer. (3) An action performed on one or more data items, such as adding, multiplying, comparing, or moving.

**operational environment.** (1) A summation of all of the Toshiba-supplied basic functions and the user programs that can be executed by the store controller to enable the devices in the system to perform specific operations. (2) The collection of Toshiba-supplied controller data and user programs, plus lists, tables, control blocks, and files that reside in a controller and control its operation. (3) The physical environment (for example: temperature, humidity, layout, or power requirements) that is needed for proper machine performance.

**operator.** (1) A symbol that represents the action being performed in a mathematical operation. (2) A person who operates a machine.

**operator message.** A message from the operating system or a program telling the operator to perform a specific function or informing the operator of a specific condition within the system, such as an error condition.

**optical character recognition (OCR).** The machine identification of printed characters through the use of light-sensitive devices.

**option.** (1) A specification in a statement, a selection from a menu, or a setting of a switch, that may be used to influence the execution of a program. (2) A hardware or software function that may be selected or enabled as part of a configuration process. (3) A piece of hardware (such as a network adapter) that can be installed in a device to modify or enhance device function.

**OS.** Operating system.

**OS/2.** Operating System/2.

**OSI.** Open Systems Interconnect.

**OS/VS.** Operating System/Virtual Storage.

**owner.** In relation to files, an owner is the user that creates the file and therefore has complete access to the file.

## P

**pacing.** A technique by which a receiving component controls the rate of transmission by a sending component to prevent overrun or congestion.

**packet.** (1) In data communication, a sequence of binary digits, including data and control signals, that is transmitted and switched as a composite whole. (2) Synonymous with *data frame*. Contrast with *frame*.

**packet assembler/disassembler (PAD).** A functional unit that enables data terminal equipments (DTEs) not equipped for packet switching to access a packet switched network.

**packing.** Method of conserving disk storage space by stripping the high-order nibbles from ASCII numerals and storing the remaining low-order nibbles two to a byte.

**PAD.** Packet assembler/disassembler.

**page.** (1) The portion of a panel that is shown on a display surface at one time. (2) To move back and forth among the pages of a multiple-page panel. See also *scroll*. (3) In a virtual storage system, a fixed-length block that has a virtual address and is transferred as a unit between main storage and auxiliary storage.

**panel.** The complete set of formatted information that appears in a single display on a visual display unit.

**parallel port.** (1) A port that transmits the bits of a byte in parallel along the lines of the bus, one byte at a time, to an I/O device. (2) On a personal computer, it is used to connect a device that uses a parallel interface, such as a dot matrix printer, to the computer. Contrast with *serial port*.

**parameter.** (1) A name in a procedure that is used to refer to an argument passed to that procedure. (2) A variable that is given a constant value for a specified application and that may denote the application. (3) An item in a menu or for which the user specifies a value or for which the system provides a value when the menu is interpreted. (4) Data passed between programs or procedures.

**parity (even).** A condition when the sum of all of the digits in an array of binary digits is even.

**parity (odd).** A condition when the sum of all of the digits in an array of binary digits is odd.

**partner.** See *conversation partner*.

**partner terminal.** The term used to describe the relationship of a Mod 1 terminal and Mod 2 terminal when they are attached to each other.

**password.** In computer security, a string of characters known to the computer system and a user, who must specify it to gain full or limited access to a system and to the data stored within it.

**path.** (1) Reference that specifies the location of a particular file within the various directories and subdirectories of a hierarchical file system. (2) In a network, any route between any two nodes. (3) The route traversed by the information exchanged between two attaching devices in a network. (4) A command in IBM DOS and IBM OS/2 that specifies directories to be searched for commands or batch files that are not found by a search of the current directory.

**PCI DSS.** Payment Card Industry Data Security Standards.

**peer node.** Any *other* SNA type (2.1) node (another 4680/4690 store controller, AS/400, or others).

**permanent virtual circuit (PVC).** A virtual circuit that has a logical channel permanently assigned to it at each data terminal equipment (DTE). A call establishment protocol is not required.

**personal computer (PC).** A desk-top, free-standing, or portable microcomputer that usually consists of a system unit, a display, a keyboard, one or more diskette drives, internal fixed-disk storage, and an optional printer. PCs are designed primarily to give independent computing power to a single user and are inexpensively priced for purchase by individuals or small businesses.

Examples include the various models of the IBM Personal Computers, and the IBM Personal System/2 computer.

**personal identification number (PIN).** A numeric identification code assigned to a customer to protect funds and data from unauthorized users.

**physical link.** In an MVS/VS multisystem environment, the actual hardware connection between two systems. Contrast with *logical link*.

**physical unit (PU).** In SNA, the component that manages and monitors the resources of a node, such as attached links and adjacent link stations, as requested by a system services control point (SSCP) through an SSCP-SSCP session.

**pipe.** A sequential file in a memory buffer that is used to pass messages from one program to another.

**PLD.** Power line disturbance.

**plug.** (1) A connector for attaching wires from a device to a cable, such as a store loop. A plug is inserted into a receptacle or plug. (2) To insert a connector into a receptacle or socket.

**point-of-sale terminal.** A unit that provides point-of-sale transaction, data collection, credit authorization, price look-up, and other inquiry and data entry functions.

**point-to-point line.** A switched or nonswitched telecommunication line that connects a single remote station to a computer. Contrast with *multipoint line*.

**polling.** (1) Interrogation of devices for purposes such as to avoid contention, to determine operational status, or to determine readiness to send or receive data. (2) In data communication, the process of inviting data stations to transmit, one at a time. The polling process usually involves the sequential interrogation of several data stations.

**polling characters (address).** A set of characters specific to a terminal and the polling operation; response to these characters indicates to the computer whether the terminal has a message to enter.

**port.** (1) An access point for data entry or exit. (2) A connector on a device to which cables for other devices such as display stations and printers are attached. Synonymous with *socket*.

**post.** (1) To affix to a usual place. (2) To provide items such as return code at the end of a command or function. (3) To define an appendage routine. (4) To note the occurrence of an event.

**POST.** Power-On Self Test.

**power line disturbance (PLD).** Interruption or reduction of electrical power.



**Power-On Self Test (POST).** A series of diagnostic tests that are run automatically each time the computer's power is switched on.

**presentation space (PS).** In 3270 emulation, the image of the 3270 screen data that is held in random access memory. This screen appears on the store controller or the terminal display when 3270 emulation is used in operator console mode; it is the virtual screen for applications using the 3270 emulator API. The presentation space is fixed as 24 lines of 80 characters on the display.

**primary adapter.** In a personal computer that is used on a LAN and that supports installation of two network adapters, the adapter that uses standard (or default) mapping between adapter shared RAM, adapter ROM, and designated computer memory segments. The primary adapter is usually designated as adapter 0 in configuration parameters. Contrast with *alternate adapter*.

**primary application.** A program that controls the normal operating environment of your store (for example, programs that provide sales support).

**primary store controller.** The store controller designated to control the store loop under normal conditions.

**prime version.** The version of a file to which updates are made. The prime version of a file may be maintained on either the Master Store Controller or the File Server. Copies of the prime version, called image versions, are distributed to other store controllers.

**printout.** Any printed document produced by a point-of-sale terminal printer or by some other printer.

**problem determination.** The process of determining the source of a problem as being a program component, a machine failure, a change in the environment, a common-carrier link, a user-supplied device, or a user error.

**procedure.** (1) A set of related control statements that cause one or more programs to be performed. (2) In a programming language, a block, with or without formal parameters, whose execution is invoked by means of a procedure call. (3) A set of instructions that gives a service representative a step-by-step procedure for tracing a symptom to the cause of failure.

**processor.** In a computer, a functional unit that interprets and executes instructions.

**Programmable Store System (PSS).** A store system, such as the Toshiba Store System, that can be programmed to perform user-determined functions.

**prompt.** A character or word displayed by the operating system to indicate that it is ready to accept input.

**protected field.** A display field that the terminal operator cannot add to, modify, or erase using the keyboard. Contrast with *input field* and *unprotected field*.

**protocol.** (1) A set of semantic and syntactic rules that determines the behavior of functional units in achieving communication. (2) In SNA, the meanings of and the sequencing rules for requests and responses used for managing the network, transferring data, and synchronizing the states of network components. (3) A specification for the format and relative timing of information exchanged between communicating parties.

**PS.** Presentation space.

**PSS.** Programmable Store System.

**PU.** Physical unit.

**public switched (telephone) network (PSN).** A telephone network that provides lines and exchanges to the public. It is operated by the communication common carriers in the USA and Canada, and by the PTT Administrations in other countries.

**PVC.** Permanent virtual circuit.

## Q

**queue.** A line or list formed by items in a system waiting for service; for example, tasks to be performed or messages to be transmitted in a message routing system.

## R

**RAM.** Random access memory.

**RAM disk.** Synonym for *virtual drive*.

**RAM paging.** A technique that allows the computer software to access all of the RAM on adapters that contain 64 KB of RAM, without having to map the entire shared RAM into the computer's memory map. The shared RAM on the adapter is paged into the computer's memory map one 16 KB page at a time.

**random access.** An access mode in which specific logical records are obtained from or placed into a mass storage file in a nonsequential manner.

**random access memory (RAM).** A computer's or adapter's volatile storage area into which data may be entered and retrieved in a nonsequential manner.

**RCMS.** Remote change management server.

**read.** To acquire or to interpret data from a storage device, from a data medium, or from another source.

**read-only memory (ROM).** A computer's or adapter's storage area whose contents cannot be modified by the user except under special circumstances.

**real storage.** The main storage in an virtual storage system. Contrast with *virtual storage (VS)*.

**receive.** To obtain and store information transmitted from a device.

**record.** A collection of related items of data, treated as a unit; for example, in stock control, each invoice could constitute one record. A complete set of such records may form a file.

**reference diskette.** A diskette shipped with the point-of-sale equipment. The diskette contains code and files used for configuration of options and for hardware diagnostic testing.

**remote change management server (RCMS).** The Toshiba Store System function that interfaces with the host DSX program for file transmission.

**remote program.** The program at the other end of a conversation with respect to the reference program. Contrast with *local program*.

**remote program load (RPL).** A function provided by adapter hardware components and software that enables one computer to load programs and operating systems into the memory of another computer, without requiring the use of a diskette or fixed disk at the receiving computer.

**remove.** (1) To take an attaching device off a network. (2) To stop an adapter from participating in data passing on a network.

**response.** The information the network control program sends to the access method, usually in answer to a request received from the access method. (Some responses, however, result from conditions occurring within the network control program, such as accumulation of error statistics.)

**retry.** In data communication, sending the current block of data a prescribed number of times or until it is entered correctly and accepted.

**return code.** (1) A value (usually hexadecimal) provided by an adapter or a program to indicate the result of an action, command, or operation. (2) A code used to influence the execution of succeeding instructions. (3) A value established by the programmer to be used to influence subsequent program action. This value can be printed as output or loaded in a register.

**ring network.** A network configuration in which a series of attaching devices is connected by unidirectional transmission links to form a closed path. A ring of an Token-Ring Network is referred to as a LAN segment or as a token-ring network segment.

**ring segment.** Any section of a ring that can be isolated (by unplugging connectors) from the rest of the ring. A segment can consist of a single lobe, the cable between access units, or a combination of cables, lobes, and/or access units. See *cable segment*, *LAN segment*.

**ring station.** A station that supports the functions necessary for connecting to the LAN and for operating with the token-ring protocols. These include token handling, transferring copied frames from the ring to the using node's storage, maintaining error counters, observing medium access control (MAC) sublayer protocols (for address acquisition, error reporting, or other duties), and (in the full-function native mode) directing frames to the correct data link control (DLC) link station.

**ring status.** The condition of the ring.

**RIPL.** Remote IPL.

**RMA.** Remote Management Agent.

**ROM.** Read-only memory.

**root directory.** Highest or base level directory in a hierarchical file system. Subdirectories branch off of the root directory.

**router.** An attaching device that connects two LAN segments, which use similar or different architectures, at the Reference Model network layer. Contrast with *bridge* and *gateway*.

**routing.** (1) The assignment of the path by which a message will reach its destination. (2) The forwarding of a message unit along a particular path through a network, as determined by the parameters carried in the message unit, such as the destination network address in a transmission header.

**RPL.** Remote program load.

## S

**SAA.** Systems Application Architecture.

**SABM.** Set asynchronous balanced mode.

**satellite.** (1) A computer that is under the control of another computer and performs subsidiary operations. (2) An offline auxiliary computer. (3) A Toshiba point-of-sale terminal under the control of a master terminal.

**SBCS.** Single-byte character set.

**scan.** To pass an item over or through the scanner so that the encoded information is read. See also *wandering*.

**scanner.** A device that examines the bar code on merchandise tickets, credit cards, and employee badges and generates analog or digital signals corresponding to the bar code.

**scroll.** To move all or part of the display image vertically or horizontally to display data that cannot be observed within a single display image. See also *page (2)*.

**SDLC.** Synchronous Data Link Control.

**SDLC link.** A data link over which communications are conducted using the Synchronous Data Link Control (SDLC) discipline.

**secondary application.** A user-written program that is designed to operate with operator intervention.

**sector.** A 512-byte area of the control unit diskette, the amount of data that is transferred at one time to or from the diskette.

**segment.** See *cable segment*, *LAN segment*, *ring segment*.

**serial port.** On personal computers, a port used to attach devices such as display devices, letter-quality printers, modems, plotters, and pointing devices such as light pens and mice; it transmits data one bit at a time. Contrast with *parallel port*.

**server.** (1) A device, program, or code module on a network dedicated to providing a specific service to a network. (2) On a LAN, a data station that provides facilities to other data stations. Examples are a file server, print server, and mail server.

**session.** (1) A connection between two application programs that allows them to communicate. (2) In SNA, a logical connection between two network addressable units that can be activated, tailored to provide various protocols, and deactivated as requested. (3) The data transport connection resulting from a call or link between two devices. (4) The period of time during which a user of a node can communicate with an interactive system, usually the elapsed time between log on and log off. (5) In network architecture, an association of facilities necessary for establishing, maintaining, and releasing connections for communication between stations.

**session group.** In System/36 advanced program-to-program communication, a number of sessions managed as a unit.

**set asynchronous balanced mode (SABM).** In communications, a data link control command used to establish a data link connection with the destination in asynchronous balanced mode. See also *asynchronous balanced mode (ABM)*.

**shared RAM.** Random access memory on an adapter that is shared by the computer in which the adapter is installed.

**signal.** (1) A time-dependent value attached to a physical phenomenon for conveying data. (2) A variation of a physical quantity, used to convey data.

**sign-on.** (1) A procedure to be followed at a terminal or workstation to establish a link to a computer. (2) To begin a session at a workstation.

**single-byte character set (SBCS).** A character set in which each character is represented by a one-byte code. Contrast with double-byte character set.

**SNA.** Systems Network Architecture.

**socket.** Synonym for *port (2)*.

**source.** The origin of any data involved in a data transfer.

**SSCP.** System services control point.

**state.** See *conversation state*.

**station.** (1) A point-of-sale terminal that consists of a processing unit, a keyboard, and a display. It can also have input/output devices, such as a printer, a magnetic stripe reader or cash drawers. (2) A communication device attached to a network. The term used most often in LANs is an *attaching device* or *workstation*. (3) An input or output point of a system that uses telecommunication facilities; for example, one or more systems, computers, terminals, devices, and associated programs at a particular location that can send or receive data over a telecommunication line. See also *attaching device*, *workstation*.

**store controller.** A programmable unit in a network used to collect data, to direct inquiries, and to control communication within a point-of-sale system.

**store loop.** In the Toshiba Store System, a cable over which data is transmitted between the store controller and the point-of-sale terminals.

**Store Loop Adapter.** A hardware component used to connect the loop to a store controller.

**subarea node.** An SNA type 5 node (a host processor), which will control all communications with the store controller.

**subdirectory.** Any level of file directory lower than the root directory within a hierarchical file system.

**subordinate store controller.** A store controller that receives copies of all system compound files and may also receive all application compound files.

**supervisory (S) frame.** A frame in supervisory format used to transfer supervisory control functions. See also *information frame*, *unnumbered frame*.

**SVC.** Switched virtual circuit.

**switch.** On an adapter, a mechanism used to select a value for, enable, or disable a configurable option or feature.

**switched line.** A telecommunication line in which the connection is established by dialing. Contrast with *nonswitched line*.

**switched virtual circuit (SVC).** A virtual circuit that is requested by a virtual call. It is released when the virtual circuit is cleared.

**symbolic destination name.** Variable corresponding to an entry in the side information.

**synchronous.** (1) Pertaining to two or more processes that depend upon the occurrence of a specific event such as a common timing signal. (2) Occurring with a regular or predictable timing relationship.

**Synchronous Data Link Control (SDLC).** A discipline conforming to subsets of the Advanced Data Communication Control Procedures (ADCCP) of the American National Standards Institute (ANSI) and High-level Data Link Control (HDLC) of the International Organization for Standardization, for managing synchronous, code-transparent, serial-by-bit information transfer over a link connection. Transmission exchanges may be duplex or half-duplex over switched or nonswitched links. The configuration of the link connection may be point-to-point, multipoint, or loop.

**system.** In data processing, a collection of people, machines, and methods organized to accomplish a set of specific functions. See also *data processing system* and *operating system*.

**system board.** In a system unit, the main circuit board that supports a variety of basic system devices, such as a keyboard or a mouse, and provides other basic system functions.

**system configuration.** A process that specifies the devices and programs that form a particular data processing system.

**Systems Application Architecture (SAA).** An architecture developed by IBM that consists of a set of selected software interfaces, conventions, and protocols, and that serves as a common framework for application development, portability, and use across different Toshiba hardware systems.

**system services control point (SSCP).** In SNA, the focal point within an SNA network for managing the configuration, coordinating network operator and problem determination requests, and providing directory

support and other session services for end users of the network. Multiple SSCPs, cooperating as peers, can divide the network into domains of control, with each SSCP having a hierarchical control relationship to the physical units (PUs) and logical units (LUs) within its domain.

**Systems Network Architecture (SNA).** The description of the logical structure, formats, protocols, and operational sequences for transmitting information units through, and controlling the configuration and operation of, networks. **Note:** The layered structure of SNA allows the ultimate origins and destinations of information, that is, the end users, to be independent of, and unaffected by, the specific SNA network services and facilities used for information exchange.

## T

**task.** A basic unit of work.

**TCC Network.** A system in which the terminals and controllers communicate using either a store loop, a token-ring or an Ethernet.

**telephone twisted pair.** One or more twisted pairs of copper wire in the unshielded voice-grade cable commonly used to connect a telephone to its wall jack. Also referred to as “unshielded twisted pair” (UTP).

**tender.** Money, checks, coupons, or trading stamps used as payment for merchandise or service.

**terminal.** In data communication, a device, usually equipped with a keyboard and a display, capable of sending and receiving information over a communication channel.

**terminal number.** A number assigned to a terminal to identify it for addressing purposes.

**threshold.** (1) A level, point, or value above which something is true or will take place and below which it is not true or will not take place. (2) In IBM bridge programs, a value set for the maximum number of frames that are not forwarded across a bridge due to errors, before a “threshold exceeded” occurrence is counted and indicated to network management programs. (3) An initial value from which a counter is decremented from an initial value. When the counter reaches zero or the threshold value, a decision is made and/or an event occurs.

**till.** A tray in the cash drawer of the point-of-sale terminal, used to keep the different denominations of bills and coins separated and easily accessible.

**token.** A sequence of bits passed from one device to another on the token-ring network that signifies permission to transmit over the network. It consists of a starting delimiter, an access control field, and an end delimiter. The frame control field contains a token bit



that indicates to a receiving device that the token is ready to accept information. If a device has data to send along the network, it appends the data to the token. When data is appended, the token then becomes a frame. See *frame*.

**token-ring.** A network with a ring topology that passes tokens from one attaching device (node) to another. A node that is ready to send can capture a token and insert data for transmission.

**token-ring network.** (1) A ring network that allows unidirectional data transmission between data stations by a token-passing procedure over one transmission medium so that the transmitted data returns to and is removed by the transmitting station. The Token-Ring Network is a baseband LAN with a star-wired ring topology that passes tokens from network adapter to network adapter. (2) A network that uses a ring topology, in which tokens are passed in a circuit from node to node. A node that is ready to send can capture the token and insert data for transmission. (3) A group of interconnected token-rings.

**TP.** Transaction program.

**trace.** (1) A record of the execution of a computer program. It exhibits the sequences in which the instructions were executed. (2) A record of the frames and bytes transmitted on a network.

**transaction.** (1) The process of recording item sales, processing refunds, recording coupons, handling voids, verifying checks before tendering, and arriving at the amount to be paid by or to a customer. The receiving of payment for merchandise or service is also included in a transaction. (2) In an SNA network, an exchange between two programs that usually involves a specific set of initial input data that causes the execution of a specific task or job. Examples of transactions include the entry of a customer's deposit that results in the updating of the customer's balance, and the transfer of a message to one or more destination points.

**transaction program (TP).** A program that processes transactions in or through a logical unit (LU) type 6.2 in an SNA network. Application transaction programs are end users in an SNA network; they process transactions for service transaction programs and for other end users. Service transaction programs are Toshiba-supplied programs that typically provide utility services to application transaction programs.

**transmission.** The sending of data from one place for reception elsewhere.

**transmit.** To send information from one place for reception elsewhere.

**twisted pair.** A transmission medium that consists of two insulated conductors twisted together to reduce noise.

**typematic.** A keyboard button that will continue to enter characters or repeat its function as long as the button is held down.

## U

**uninterruptible power supply.** A buffer between utility power or other power source and a load that requires uninterrupted, precise power.

**universal product code (UPC).** An encoded number that can be assigned to and printed on or attached to an article of merchandise for scanning.

**universal serial bus.** An industry standard that makes it easy to expand PC functionality. The USB is a 12-Mbps serial bus designed to replace almost all low-to-medium speed connections to peripheral devices such as keyboards, mice, and printers.

**unnumbered acknowledgment.** A data link control (DLC) command used in establishing a link and in answering receipt of logical link control (LLC) frames.

**unnumbered (U) frame.** A frame in unnumbered format, used to transfer unnumbered control functions. See also *information frame*, *supervisory frame*.

**unprotected field.** A display field that the terminal operator can add to, modify, or erase using the keyboard. Contrast with *protected field*.

**UPC.** Universal product code.

**UPS.** Uninterruptible power supply.

**USB.** universal serial bus

**user.** (1) Category of identification defined for file access protection. (2) A person using a program or system.

**user exit.** A point in a Toshiba-supplied program at which a user-written program may be given control.

**utility program.** (1) A computer program in general support of the processes of a computer; for instance, a diagnostic program, a trace program, a sort program. (2) A program designed to perform an everyday task such as copying data from one storage device to another.

## V

**variable.** (1) A named entity that is used to refer to data and to which values can be assigned. Its attributes remain constant, but it can refer to different values at different times. (2) In computer programming, a character or group of characters that refers to a value and, in the execution of a computer program, corresponds to an address. (3) A quantity that can assume any of a given set of values.

**version.** A separate Toshiba-licensed program, based on an existing Toshiba-licensed program, that usually has significant new code or new function.

**VFD.** Vacuum fluorescent display.

**VFS.** virtual file system.

**video display.** (1) An electronic transaction display that presents visual information to the point-of-sale terminal operator and to the customer. (2) An electronic display screen that presents visual information to the display operator.

**virtual circuit.** Synonym for *virtual connection*.

**virtual connection.** (1) A connection between two nodes on the network that is established using the transport layer and provides reliable data between nodes. (2) A logical connection established between two data terminal equipment (DTE) devices. Synonymous with *virtual circuit*.

**virtual drive.** Computer memory used as if it were a direct access storage device. Synonym for *RAM disk*.

**virtual file system (VFS).** Within 4690 OS V2, the virtual file system is used to provide support for long file names by creating two virtual drives that support file names greater than eight characters in length.

**virtual machine (VM).** A functional simulation of a computer and its associated devices. Each virtual machine is controlled by a suitable operating system, for example, a conversational monitor system. VM controls concurrent execution of multiple virtual machines on one host computer.

**virtual storage (VS).** (1) The storage space that may be regarded as addressable main storage by the user of a computer system in which virtual addresses are mapped into real addresses. The size of virtual storage is limited by the addressing scheme of the computer system and by the amount of auxiliary storage available, not by the actual number of main storage locations. (2) Addressable space that is apparent to the user as the processor storage space, from which the instructions and the data are mapped into the processor storage locations. Contrast with *real storage*.

**VM.** Virtual machine.

**VS.** Virtual storage.

## W

**wand.** A commercially available device used to read information encoded on merchandise tickets, credit cards, and employee badges.

**wanding.** Passing the tip of the wand reader over information encoded on a merchandise ticket, credit card, or employee badge.

**wideband.** Synonym for *broadband*.

**work file.** A file that is both created and deleted in the same job.

**workstation.** (1) An I/O device that allows either transmission of data or the reception of data (or both) from a host system, as needed to perform a job: for example, a display station or printer. (2) A configuration of I/O equipment at which an operator works. (3) A terminal or microcomputer, usually one connected to a mainframe or network, at which a user can perform tasks.

## X

**XID.** Exchange identification.

**X.21.** In data communication, a recommendation of the CCITT that defines the interface between data terminal equipment (DTE) and public data networks for digital leased and circuit switched synchronous services.

**X.21 bis.** In data communication, an interim specification of the CCITT that defines the connection of data terminal equipment (DTE) to an X.21 (public data) network using V-series interchange circuits such as those defined by CCITT V.24 and CCITT V.35.

**X.25.** A CCITT Recommendation that defines the physical level (physical layer), link level (data link layer), and packet level (network layer), of the OSI Reference Model. An X.25 network is an interface between data terminal equipment (DTE) and data circuit-terminating equipment (DCE) operating in the packet mode, and connected to public data networks by dedicated circuits. X.25 networks use the connection-mode network service.

---

# Index

## Special characters

/\* \*/ 171

\$ option, LINK86 214

\$C (command) option, LINK86 214

\$D option, LINK86 214

\$L (library) option, LINK86 214

\$M 205

\$M (map) option, LINK86 214

\$N (line number) option, LINK86 214

\$O 205

\$O (object) option, LINK86 215

\$S (symbol) option, LINK86 215

\$X 205

## Numerics

286 executable load module 207

286 file type 207

2x20 displays 39

accessing 40

characteristics of 39

clearing 40

current character location 40

customizing alphanumeric display character set 41

customizing LCD display character set 41

customizing VFD II display character set 41

determining information 42

display character sets 41

example application of 43

programming hints 43

programming tips for 39

reading from 42

writing characters to 40

4610 printer driver 120

4610 printer, characteristics 120

4683-xx1 terminal storage retention 162

4683-xx2 terminal storage retention 162

4689 native mode support 130

4689 printer driver 127

4689 printer, characteristics 127

4690 Java-BASIC API introduction 291

## A

abbreviations for LIB86 command-line options 201

access

modes 335

random 336

sequential 336

accessing

2x20 displays 40

cash drawer driver 61

coin dispenser driver 63

distributed files in terminal applications 24

files 30

files in other directories 205

- accessing *(continued)*
  - files on LAN system 27
  - files using node name 27
  - I/O processor 72
  - key file utility 177
  - magnetic stripe reader 89
  - pipe 337
  - printer 100
  - printer 4610 122
  - printer 4689 128
  - printer model 3 107
  - printer model 4/4A 107
  - RAM disk files from controller applications 315
  - RAM disk files from terminal applications 315
  - scale driver 139
  - serial I/O communications driver 141
  - shopper display 45
  - tone driver 146
  - totals retention driver 148
  - totals retention driver in direct mode 148
  - video display 50
- active controllers on network, get all 305, 371, 503
- active user, disconnect 308, 376
- active user, sign off 308, 376
- adding
  - a table using input sequence table utility 199
  - operator record to authorization file 283
- ADDITIONAL parameter for LINK86 211
- additional products 9
- ADX files
  - ADXCSW0L, command formats for 228
  - ADXCSW0L, parameters for 228
- ADX\_CABORT\_PROS 384
- ADX\_CAUTH 364
- ADX\_CCHANGE\_ATTRIB 356
- ADX\_CCLOSE\_FILE 351
- ADX\_CCLOSE\_LS 364
- ADX\_CCREATE\_FILE 348
- ADX\_CCREATE\_KFILE 338
- ADX\_CCREATE\_POSFILE 346
- ADX\_CCRYPT 366
- ADX\_CDELETE\_FILE 354
- ADX\_CDELETE\_KREC 346
- ADX\_CERROR 383
- ADX\_CEXIT\_PROS 385
- ADX\_CFILES 357, 359, 360
- ADX\_CGET\_STAT 364
- ADX\_CLOCK\_REC 353
- ADX\_CMEM\_FREE 386
- ADX\_CMEM\_GET 385
- ADX\_COPEN\_FILE 349
- ADX\_COPEN\_KFILE 342
- ADX\_COPEN\_LINK 364
- ADX\_COPEN\_SESS 364
- ADX\_CPRS\_CREATE 361
- ADX\_CPRS\_CWRITE 363
- ADX\_CPRS\_INIT 361
- ADX\_CPRS\_READ 361
- ADX\_CPRS\_WRITE 362
- ADX\_CPRTHEX 388
- ADX\_CREAD\_HOST 364

- ADX\_CREAD\_KREC 343, 344
- ADX\_CREAD\_REC 351
- ADX\_CRENAME\_FILE 355
- ADX\_CSEEK\_PTR 356
- ADX\_CSEND\_REQ 364
- ADX\_CSERVE 367
- ADX\_CSTARTP 382
- ADX\_CTIMER\_SET 386
- ADX\_CWAIT 387
- ADX\_CWRITE\_HOST 364
- ADX\_CWRITE\_KREC 345
- ADX\_CWRITE\_REC 352
- ADX\_TCOPYF 444
- ADX\_TCRYPT 444
- ADX\_TDIR 443
- ADX\_TERROR 442
- ADX\_TSERVE 436
- ADXAUTH function 283
  - adding operator record 283
  - changing authorization record 283
  - changing password 283
  - deleting operator record 283
  - example of 283
- ADXCOPYF 315, 332
- ADXCRIPT 285
- ADXC50L 242
- ADXCSE0L 320
- ADXCSEUR 320
- ADXC5K0L 179
- ADXD1R 322
- ADXERROR 312
- ADXEXIT 321
- ADXF1LES routines
  - despool 319
  - overview 317
  - restrict 317
  - unrestrict 318
- ADX1L10F.DAT 239
- ADXNSXZL 269
- ADXP1K0:, JavaPOS logical device 590
- ADXP1S0:, JavaPOS logical device 590
- ADXPSTAR 281
- ADXSERCL 321
- ADXSERVE 294
- ADXSTART 281
- ADXSTART function 280
- AJ command (print spooler) 224
- alarm for cash drawer
  - controlling 61
  - obtaining status 61
- alerts, audible alarm 159
- algorithm
  - alternate hashing 183
  - folding 183
  - modulo check 71
  - polynomial hashing 189
  - specifying for files 184
  - UPC/EAN modulo check 71
  - user-defined modulo-check 70
- allocate memory 385
- allocating space for files 29

- alternate hashing algorithm 183
- ANDisplayHandler Java class 548
  - methods 549
    - clear() 549
    - close() 549
    - getColumn() 550
    - getDisplayType() 550
    - getRow() 550
    - locate() 549
    - open() 549
    - read() 549
    - write() 550
- ANDisplayHandlerException Java class 557
  - constructors 557
    - ANDisplayHandlerException() 557
    - ANDisplayHandlerException(String) 557
  - method
    - getMessage() 558
    - getReturnCode() 558
    - setDeviceOfflineRC() 558
    - setHardwareFailureRC() 558
    - setInvalidCursorPositionRC() 558
  - methods 558
- appending an existing library 203
- application action, error codes 163
- application environment, understanding 277
- application errors 156
- application events, logging 312
- application functions, printer 100
- application functions, printer 4610 122
- application functions, printer model 3 or 4/4A 107
- application logical file names 21
- application priorities
  - description of 280
  - file access priorities 280
- application program interface
  - API (DOS) 171
  - API (OS/2) 170
  - application program interface (DOS) 171
  - application program interface (OS/2) 170
- application responses to errors 165
- application service errors, CPI 452
- application services
  - ADX\_CC\_CREATE\_KFILE 338
  - ADX\_CFILES 357
  - ADX\_CFILES Despool 360
  - ADX\_CFILES Restrict 357
  - ADX\_CFILES UnRestrict 359
  - ADX\_CSERVE 367
  - ADXCOPYF 315, 332
  - ADXDIR 322
  - ADXEXIT 321
  - ADXFILES 317
  - ADXFILES Despool 319
  - ADXFILES restrict 317
  - ADXFILES Unrestrict 318
  - ADXPSTAR 281
  - ADXSERCL 321
  - ADXSTART 280
  - ASCII to EBCDIC, converting 303
  - canceling files 317, 357

- application services (*continued*)
  - closing ADXSERVE using ADXSERCL 321
  - copying files 315, 332
  - disable controller programmable power 301, 503
  - disconnect active user 376, 511
  - displaying application status 302
  - displaying background application status 302
  - EBCDIC to ASCII, converting 303
  - enable controller programmable power 301, 508
  - getting configured controllers on the network 303, 503
  - getting controller ID for a terminal 303, 503, 505
  - getting disk free space 303
  - listing files 322
  - power off a local machine (controller or terminal) 301, 506, 528
  - power off a remote controller 300, 506
  - power off a remote terminal 300, 507
  - requesting 293
  - requesting (ControllerApplicationServices) 502
  - restart a background application 301
  - setting error level value files 321
  - sign off active user 376, 510
  - starting a background application 280
  - stop application with message 302
  - store controller unique 321
  - using 293
  - wait for keyboard and mouse activity 375
  - wait for keyboard and mouse inactivity 375, 510
  - wait for keyboard or mouse activity 510
- application size 278
- application status 367, 368
- application status information 296
- application-to-application communication between terminals and the store controller 286
- Application, start Java 311
- Application, stop Java 312
- applications
  - background 277
  - chaining 313
  - displaying status 302
  - executing DOS 391
  - interactive 277
  - logical file names 21
  - operator interactive 277
  - primary 277
  - priorities 280
  - restart background 301
  - secondary 277
  - size 278
  - stopping with message 302
  - temporary background 277
- apply software maintenance 242
- AQ command (activate queue) 224
- architecture 137
- ASCII to EBCDIC, converting 303, 370
- ASCII, character sets 277
- at close, distributed file 341
- attributes 356
- attributes, changing 356
- audible alarm
  - alerts 159
  - building a message list 159
  - deactivating 161

- audible alarm (*continued*)
  - for system messages 159
  - LAN use 160
  - setting 159
  - setting through RCP 159
  - using 160
- authorization
  - operator 364
  - operators, using system authorization file 282
  - passwords 364
  - record ID 282
- Auto Sign Off
  - CD/DVD media
    - eject 310, 382, 511
  - disconnect active user 6
    - C language 376
    - CBASIC 308
    - Java 511
  - programming 5
  - sign off active user 5
    - C language 376
    - CBASIC 308
    - Java 510
  - wait for keyboard and mouse activity
    - C language 375
    - CBASIC 308
    - Java 510
  - wait for keyboard and mouse inactivity 5
    - C language 375
    - CBASIC 308
    - Java 510

## B

- BACKGRND parameter 278, 382, 383
- background applications
  - definition of 277
  - displaying status 278, 302, 369
  - permanent 277
  - starting 280, 281, 382
  - temporary 277
  - types of 277
- backup
  - data 9
  - store controller 9
- bar code reader
  - label format 70
- BASIC application, store closed query 320
- BAT file, creating 266
- battery return program 758
- big memory model, defined 278
- BIOS calls 394
- bit numbering 334
- boot archives
  - JavaTOF 261
- buffering the journal 111
- building logical names table 23
- building message list, audible alarm 159



## C

- C applications, terminal 333
- C language interfaces, 16-bit 334
- C language interfaces, 32-bit 334
- C Programming Interface for 4690 terminals
  - 16-bit libraries 397
  - 32-bit libraries 397
  - access modes 400
  - asynchronous error interrupt handling 398
  - close a keyed file 410
  - closing a file 406
  - closing a pipe 406
  - common file functions 401
  - compiling 399
- CPI device interface 415
  - clearing the display 419
  - closing a device 418
  - defining asynchronous error handler 421
  - flush pending writes to printer 427
  - get total retention offset 423
  - getting device status 419
  - loading state tables 420
  - locking a device 420
  - opening a device 415
  - opening I/O processor 416
  - opening serial I/O session 417
  - positioning the cursor 418
  - read direct from totals retention 425
  - reading data from fiscal printer 428
  - reading from a device 423
  - set totals retention offset 423
  - setting device status 419
  - unlocking a device 420
  - wait for event completion 425
  - write bitmap data to printer 427
  - write direct to totals retention 427
  - writing to a device 425
- CPI error codes 450
  - application service errors 452
  - device access errors 451
  - file access errors 451
  - general errors 450
  - miscellaneous functions errors 453
  - pipe access errors 451
  - pipe routing services errors 452
- creating non-pos file 404
- creating pipe 404
- delete a keyed file record 411
- deleting a file 401
- error handling 398
- extended memory management (469x POS terminal) 445
  - accessing shared memory file 446
  - allocating a memory file 445
  - binary search 448
  - clearing memory file 449
  - data buffers 445
  - deleting data from memory file 449
  - error codes 449
  - error handling 445
  - freeing a memory file 446
  - gaining mutually exclusive access 446

- C Programming Interface for 4690 terminals *(continued)*
  - extended memory management (469x POS terminal) *(continued)*
    - querying available free memory 447
    - reading from memory file 447
    - releasing exclusive access 446
    - sequential search 448
    - writing data to file in ascending order 448
    - writing to memory file 447
  - file pointers 400
  - file security 400
  - file services 399, 403
  - getting file pointers 402
  - getting file size 403
  - header file 399
  - heap manager 398
  - introduction 397
  - linking 399
  - miscellaneous routines 428
    - allocating heap space 428
    - chaining to another program 429
    - converting hex to ASCII 435
    - disabling software interrupts 431
    - enabling software interrupts 431
    - freeing heap space 429
    - freeing storage 434
    - getting system date 431
    - getting system time 431
    - pack data 431
    - query available heap space 429
    - query contiguous available heap space 429
    - receiving parameters from chaining program 430
    - requesting OS memory 432
    - suspending processing 434
    - unpack data 432
    - waiting for data 434
  - open a keyed file 409
  - opening a file 405
  - opening a pipe 405
  - pipe routing services 412
    - close a pipe 413
    - closing the driver 415
    - conditional write to a pipe 414
    - creating a pipe 412
    - open the driver 413
    - reading a pipe 413
    - waiting for data in pipe 413
    - writing to a pipe 414
  - pipe services 403
  - product signature 450
  - read a keyed file 410
  - reading from a file 406
  - reading from a pipe 406
  - renaming a file 402
  - restrictions 398
  - setting file pointers 402
  - setting file security 401
  - terminal application services 436
    - ADX\_TCOPYF 444
    - ADX\_TCRYPT 444
    - ADX\_TDIR 443
    - ADX\_TERROR 442

- C Programming Interface for 4690 terminals *(continued)*
  - terminal application services *(continued)*
    - ADX\_TSERVE 436
    - copying disk files 444
    - disable IPL 440
    - disabling terminal storage retention 437
    - dump system storage 436
    - enable IPL 440
    - enable terminal storage retention 437
    - encrypting a password 444
    - get application status information 437
    - listing terminal hard disk files 443
    - listing terminal RAM disk files 443
    - logging an application event 442
    - programmable power 439
    - requesting an application service 436
    - validation of application buffers 450
    - write to a keyed file 411
    - writing a string array to sequential file 408
    - writing to a file 407
    - writing to a pipe 407
  - cable ferrite requirement 756
  - Cancel 712
  - canceling shared use of a file 317
  - canceling the shared use of a file using ADX\_CFILES 357
  - cash drawer 60
  - cash drawer driver
    - accessing 61
    - characteristics of 60
    - controlling alarm for 61
    - example of code for 62
    - obtaining alarm status 61
    - programming tips for 60
  - CashReceiptMonitor Java class 554
    - method 554
    - printData() 554
  - chaining
    - applications 313
    - directly executable modules 314
    - overlays 313, 314
    - stats from a direct file 179
    - terminal applications 314
    - threshold 192
  - ChangePassword 687
  - changing
    - authorization using ADXAUTH function 283
    - file distribution attributes 356
    - file pointer 356
    - fonts 109, 124
    - logical names table 23
    - password in operator record 283
    - tables using input sequence table utility 198
  - changing the signon screen 4
  - character sets
    - ASCII 277
    - check printing 741
    - double width 742
    - model 1 printer 103
    - model 2 printer 103
    - normal width 741

- characteristics
  - 4610 printer 120
  - 4689 printer 127
  - model 3 printer 106
  - model 4/4A printer 106
- check a file in exception logs 375
- check file server exception log 505
- check master exception log 505
- check printing
  - character sets 741
  - characteristics of, model 1 or 2 printer 101
  - example of 742
  - formatting the data for, model 1 or 2 printer 102
  - problem determination for, model 1 or 2 printer 103
  - programming considerations, model 1 or 2 printer 103
  - warning 103
- CJ command (print spooler) 224
- Class A compliance statement
  - Australia and New Zealand 752
  - China 755
  - European Union 752, 756
  - FCC (USA) 752
  - Germany 753
  - Industry Canada 752
  - Japan 756
  - Russia 754
  - Taiwan 755
- class and parameters
  - Java 2 265
- Class FlexosException constructors 478
- Class FlexosException methods 478
- Class InvalidParameterException constructor 479
- Class JavaInvocationResult 546
- Class POSPlatform 539
- classes in package com.ibm.OS4690.jiop 593
- classes in package com.ibm.OS4690.jiop.util 650
- classpath setup, terminal 263
- CLEAR key use in Input Sequence Table 67
- clearing
  - 2x20 displays 40
  - shopper display 45
  - video display 50
- close file or pipe 351
- close file stream (KeyedFile class) 495
- close keyed file 343
- close output pipe (POSPipeOutputStream) 500
- close output stream (POSPipeOutputStream) 500
- close, partial 343, 351
- closing
  - application files using TCLOSE 167
- closing ADXSERVE using ADXSERCL 321
- COBOL interfaces 334
- COBOL/2 334
- COBOL/2, 334
- CODE section 210
- code size, when exceeds available memory 278
- CODESHARED option 213
- coding tips for writing non-4680 BASIC programs 333
- coin dispenser driver
  - accessing 63
  - characteristics of 62

- coin dispenser driver *(continued)*
  - dispensing coins 63
  - example of code for 63
  - programming tips for 62
- com.ibm.OS4690.jiop 592, 593
- com.ibm.OS4690.jiop.util 592, 650
- command file, example of 240
- command formats for ADXCSW0L 228
- command options, LINK86 209
- command stacking
  - example of 58
  - restrictions using 58
  - using 57
- command-line files for LIB86 202
- command-line options 201
- commands
  - AJ 224
  - AQ 224
  - CJ 224
  - CQ 225
  - HQ 224
  - LQ 224
  - PJ 223
  - TJ 223
  - TQ 225
- commands, print spooler
  - AJ 224
  - AQ 224
  - CJ 224, 225
  - HJ 224
  - HQ 224
  - LJ 224
  - PJ 223
  - RJ 225
  - TJ 223, 225
- commands, special, processing 670
- Commit 711
- common information for input sequence table 66
- communicating between applications
  - in store controller applications 286
- communications, system 7
- compatibility, printer 106, 121
- compound file
  - creating 341
  - parameters 348, 357
- concurrent operations on the system 4
- conditional write to PRS pipe 363
- configuration considerations, RAM disks 314
- configure redirected devices to Java 540
- configured controllers, getting 370
- controller application status information 504
- controller applications
  - accessing RAM disk files from 315, 324
  - copying RAM disk files from 315
  - listing directories RAM disk file from 315
- controller, store configured on network 303, 503
- ControllerApplicationServices class, Java 502
  - methods 503
    - disableControllerProgrammablePower() 503
    - disableStorageRetention() 503
    - disconnectActiveUser() 511

ControllerApplicationServices class, Java (continued)

methods (continued)

- dumpSystemStorage() 503
- getAllActiveNetworkControllers() 503
- getConfiguredNetworkControllers() 503
- getControllerId(short) 503
- getControllerModelAndType() 504
- getControllerStatus() 504
- getLoopIDForTerminal(short) 504
- getLoopMessage(String, byte) 504
- getLoopStatus() 505
- getStoreControllerIDForTerminal(short) 505
- isController() 505
- isDumpFlatOn() 505
- isEntriesInFileServerExceptionLog() 505
- isEntriesInMasterExceptionLog() 505
- logError(char, int, int, int, byte[]) 506
- powerOffLocalMachine(String) 506
- powerOffRemoteController(String) 506
- powerOffRemoteTerminal(String) 507
- resetDumpFlag() 508
- runRSTConAllTerm() 510
- runRSTConTerm(int) 510
- setDumpFlag() 508
- setLocalControllerProgrammablePower() 508
- setStorageRetention() 508
- setTerminalSleepModeTimer(short) 508
- signoffActiveUser() 510
- startBackgroundApplication(String, byte[], String, short) 508
- waitForKeyboardMouseActivity() 510
- waitForKeyboardMouseInactivity (short waitTime) 510

controllers, get all active on network 305, 371, 503

ControllerStatusData class, Java

fields 514

- ALT\_MASTER\_ALT\_FILESERVER 514
- ALTERNATE\_FILE\_SERVER 514
- ALTERNATE\_MASTER 514
- COLOR\_DISPLAY 514
- COMMA\_CONVENTION 514
- CONSOLE\_0 514
- CONSOLE\_1 514
- CONSOLE\_2 514
- CONSOLE\_3 514
- CONSOLE\_4 515
- CONSOLE\_5 515
- CONSOLE\_6 515
- CONSOLE\_7 515
- CONSOLE\_8 515
- CONTROLLER\_NOT\_ON\_LAN 515
- CONTROLLER\_ON\_LAN 515
- D\_M\_Y\_FORMAT 515
- DAY\_MONTH\_YEAR\_FORMAT 515
- ETHERNET 515
- FILE\_SERVER 516
- HMS\_TIME\_FORMAT 516
- HOURLY\_MINUTE\_SECOND\_FORMAT 516
- LAN\_NOT\_INSTALLED 516
- M\_D\_Y\_FORMAT 516
- MASTER 516
- MASTER\_AND\_FILE\_SERVER 516
- MONOCHROME\_DISPLAY 516

- ControllerStatusData class, Java *(continued)*
  - fields *(continued)*
    - MONTH\_DAY\_YEAR\_FORMAT 516
    - PERIOD\_CONVENTION 516
    - TOKENRING 516
    - TWO\_SIGNIFICANT\_DIGITS 517
    - UNKNOWN\_DISPLAY 517
    - ZERO\_SIGNIFICANT\_DIGITS 517
  - methods 517
    - getActingControllerType() 517
    - getAssignedConsole() 517
    - getControllerId() 518
    - getDateFormat() 518
    - getDisplayType() 518
    - getLanConnectionType() 519
    - getMasterControllerId() 519
    - getMonetaryFormat() 519
    - getNumberOfDigitsAfterDecimal() 519
    - getNumberPrinterLinesPerPage() 519
    - getNumberTerminalsConfigured() 519
    - getPrinterAssociatedWithConsole() 520
    - getStoreNumber() 519
    - getTimeFormat() 520
    - isControllerOnActiveLan() 517
- converting
  - ASCII to EBCDIC 303, 370
  - EBCDIC to ASCII 303, 370
  - HEX to ASCII 388
- Copy 693
- copying
  - files 33
  - files using ADXCOPYF 315, 332
  - RAM disk files from controller applications 315
  - table, input sequence table utility 198
- copying disk files 444
- CPI extended memory management (469x POS terminal) 445
- CQ command (print spooler) 225
- Create 701
- create and open a distributed keyed file, Java class 490
- create and open a local keyed file, Java class 492
- create and open distributed keyed file Java class 493
- create and open local keyed file Java class 494
- creating
  - an empty file (batch method) 181
  - compound files 341
  - creating files 336
  - files 29, 348
  - input file 207
  - keyed file 338
  - keyed file from a direct file 179
  - library files 204
  - non-POS file or pipe 338, 348
  - pipes 336, 348
  - POS non-keyed file 346
  - PRS pipe 361
- creating a BAT file 266
- creating a library file 202
- cutter control characters 114, 125

## D

- damage from electrostatic discharge 756
- data
  - backup 9
  - DATA section 210
  - formatting for check printing, model 1 or 2 printer 102
  - heap 279
  - integrity for PLDs 37
  - receiving from the I/O processor 72
  - recovering using Desk Surface Analysis Utility 231
  - size, terminal default 279
  - stack 279
  - static 279
  - waiting for from I/O processor 73
- date, system, getting 431
- DBCS support for Java 2 657
- DBINFO option, LINK86 215
- Debug (DBG) file 207
- default
  - data size for terminals 279
  - values 210
- defining
  - input sequence 66
  - logical names 21
- defining logical file names 21
- Delete 702
- delete a record (KeyedFile class) 495
- DELETE option, LIB86 203
- deleting
  - files 30
  - files or pipes 354
  - functions of 335
  - keyed record 346
  - modules from library files 203
  - operator record to authorization file 283
  - pipes 336
- designing applications with Java 455
- designing terminal applications with C 397
- despooling
  - effects of activating alternate file server on 169
- determining printer status, 4610 124
- determining printer status, 4689 129
- determining printer status, model 3 or 4/4A 113
- device access errors, CPI 451
- device programming
  - 40-character Liquid Crystal Display (LCD) 39
  - 40-character Vacuum Florescent Display II (VFD II) 39
  - alphanumeric display 39
  - APA display 39
  - cash drawer driver 60
  - coin dispenser driver 62
  - Dual-Track Magnetic Stripe Reader 84
  - I/O processor 64
  - operator display 39
  - printer driver 99, 106, 120, 127
  - scale driver 139
  - serial I/O communications driver 141
  - shopper display 44
  - tone driver 146
  - Two-sided VFD II 39
  - video display 47



- DeviceManager 540
  - handlers 539
  - handlers and monitors 539
  - methods 541
    - createDeviceManager(int terminalID) 541
    - IOPDataAvailable() 541
    - MSRDataAvailable() 541
    - POSPrinterDataAvailabe (int nStation) 541
    - run() 541
    - setANDisplay1Handler(ANDisplayHandler) 542
    - setANDisplay2Handler(ANDisplayHandler) 542
    - setCashReceiptMonitor(CashReceiptMonitor) 542
    - setDeviceRegistrationComplete() 542
    - setExtendedIOPHandler(ExtendedIOPHandler) 542
    - setIOPHandler(IOPHandler) 542
    - setMSRHandler(MSRHandler) 542
    - setPOSPrinterHandler(POSPrnterHandler) 542
    - singleton() 541
  - monitors 540
    - CashReceiptStation 540
- DeviceManagerInterface 543
- DeviceManagerInterfaceMultiJVM 544
- direct file, description of 12
- directly executable modules, chaining 314
- directmode
  - accessing totals retention driver in 148
  - reading totals retention driver data in 148
  - writing totals retention driver data in 148
- directories of terminal applications, listing 315
- disable
  - security for files and subdirectories 35
  - terminal IPL 307
  - terminal storage retention 296, 368, 503, 527
- disable IPL 440
- disable terminal storage retention 296, 503, 527
- disabling terminal storage retention 437
- disconnect active user 376
- disconnect the active user 308
- disk
  - and file error recovery 162
  - directory 727
  - file management 335
  - getting free space 303, 369
  - media 335
  - space, total 278
- Disk Surface Analysis Utility
  - ADXCSWOL, command formats for 228
  - ADXCSWOL, parameters for 228
  - example of a command file 240
  - examples of disk recovery 232
  - file allocation table (FAT) 228
  - function of 227
  - IPL command processor 239
  - LAN, Defect in .286 File on Alt Mstr Contr 233
  - LAN, Defect in .286 File on Mstr Contr 233
  - LAN, Defect in Item Record File on Alt Mstr 236
  - LAN, Defect in Item Record File on Mstr Contr 236
  - LAN, Defect in Unallocated Space on Alt Mstr Contr 235
  - LAN, Defect in Unallocated Space on Mstr Contr 234
  - LAN, Defect Near End of TLOG on Alt FS 237
  - LAN, Defect Near End of TLOG on FS 237

- Disk Surface Analysis Utility *(continued)*
  - Non-LAN, Defect in .286 File 232
  - Non-LAN, Defect in Item Record File 235
  - Non-LAN, Defect in Unallocated Space 234
  - Non-LAN, Defect Near End of TLOG 237
  - recovering data 231
  - time frame indicators 232, 239
  - usage of 227
- dispensing coins 63
- display character sets 39
  - characteristics for 2x20 displays 39
  - customizing alphanumeric display character set 41
  - customizing LCD display character set 41
  - customizing VFD II display character set 41
  - for 2x20 displays 41
  - for shopper display 45
  - for video display 52
- Display Manager 9
- display output handling exceptions 557
- displaying
  - application status message 302, 369
  - background application status msg 302, 369
  - logical names table 23
  - restart background application in the same slot 301
  - table using input sequence table utility 198
- displays
  - 40-character Liquid Crystal Display (LCD) 39
  - 40-character Vacuum Florescent Display II (VFD II) 39
  - alphanumeric 39
  - APA 39
  - operator 39
  - shopper 44
  - Two-sided VFD II 39
  - video 47
- disposal of equipment 757
- distributed file
  - accessing in terminal applications 24
  - at close 341
  - ways to 348
- distribution errors, logging 28
- distribution exception log 161
- document
  - eject command 112, 124
  - insert station 100, 109, 124
  - insertion, manual or automatic 110
  - options 111
  - station character line lengths 113, 124
- DOS
  - executing 391
  - function calls 395
  - program memory allocation 392
- double strike printing 108
- drawer, cash 60
- dual track magnetic stripe reader driver
  - accessing 89
  - characteristics 85
  - common data characteristics 89
  - determining status of 90
  - disallowing data 90
  - example of code for 92, 93, 95
  - Magnetic Stripe Reader 84

- dual track magnetic stripe reader driver *(continued)*
  - preparing to receive data from 89
  - receiving data from 89
  - restrictions 88
  - waiting for received data 89
- dump system storage 295, 367, 436, 503, 527

## E

- EBCDIC to ASCII, converting 303, 370
- ECHO option 214
- eject CD/DVD media 310, 382, 511
- electromagnetic Interference statement
  - Russia 754
- electronic emissions notices 752
  - Australia and New Zealand 752
  - China 755
  - European Union 752, 756
  - FCC (USA) 752
  - Germany 753
  - Industry Canada 752
  - Japan 756
  - Korea 754
  - Taiwan 755
- electrostatic discharge (ESD) 756
- emphasized printing 108
- emulator messages 396
- emulator report, optional 392
- enable
  - security for files and subdirectories 35
  - terminal IPL 307
  - terminal storage retention 295, 367, 508, 529
- enable IPL 440
- enable terminal storage retention 437
- enable/disable IPL
  - disable terminal IPL 307
  - enable IPL function 307
  - using 306
- encrypting a password 444
- encrypting password 285, 366
- end of life disposal 757
- ending
  - a program 384
  - access to files 31
- enhanced passwords
  - APIs 684
  - application programming interfaces 684
  - authorization attributes 680
  - Cancel 712
  - ChangePassword 687
  - Commit 711
  - Copy 693
  - Create 701
  - Delete 702
  - error codes 681
  - GetAllAttributes 687
  - GetAllAttributesCurrentUser 688
  - GetAttribute 691
  - GetAttributes 690
  - GetAuthorizationLevel 699
  - GetCurrentUserID 685

- enhanced passwords (*continued*)
  - GetExpires 706
  - GetExpireWarning 708
  - GetGroupNumber 697
  - GetIDs 704
  - GetMinChange 709
  - GetMinPasswordLength 707
  - GetModelCount 705
  - GetModelIDs 705
  - GetMultipleCharacter 710
  - GetRecordCount 704
  - GetUserAttributes 692
  - GetUserNumber 698
  - IsAvailable 685
  - Lock 692
  - password rules 679
  - Rename 702
  - sessions 683
  - SetAttribute 695
  - SetAttributes 694
  - SetAuthorizationLevel 700
  - SetExpires 700, 706
  - SetExpireWarning 708
  - SetGroupNumber 697
  - SetMinChange 709
  - SetMinPasswordLength 707
  - SetMultipleCharacter 711
  - SetPassword 703
  - SetUserAttributes 696
  - SetUserNumber 698
  - StartSession 689
  - StartSessionCurrentUser 690
  - user ID rules 679
  - user-defined attributes 681
  - using 679
  - Validate 686
  - ValidateID 686
- Enhanced Security 679
- enhancements, JavaTOF procedural 266
- environment, operating system 3
- equipment disposal 757
- erasing
  - a table using input sequence table utility 198
  - spool file 168
- ERR function 155
- ERRF% function 155
- ERRL function 155
- ERRN function 155
- error
  - handling 138
  - recovery 138
- error codes
  - application action 163
  - description 163
  - print spooler 225
- error functions 153
- error logging
  - ADXERROR 312
  - error log 155
  - system log 155
- error recovery 153

- error statements 153
- ERRORLEVEL test 216, 219
- errors
  - application 156
  - application responses 165
  - hardware, store controller 156
  - hardware, terminal 156
  - I/O devices 165
  - LINK86 messages 734
  - logging 155
  - POSTLINK, messages 732
  - recovery from 153
  - setting error level value files 321
  - store controller 156
  - system 156
  - terminal 156
- errors and exceptions, redirection 557
- European Union battery recycling statement 758
- event completion, waiting for 387
- event log access
  - date format 158
  - description 157
  - record formats 158
  - time format 159
- event logging, ADXERROR 312
- example of 4680 BASIC program 173
- example of a command file 240
- example of C/2 C-language program 171
- example of DOS C-language program 172
- exception handling, run-time 672
- ExceptionEvent Java class 652
  - constructors 652
    - ExceptionEvent(Object, Throwable) 652
    - ExceptionEvent(Object) 652
  - method 652
    - getThrowable() 652
- ExceptionListener Java class 652
  - method 652
    - exception(ExceptionEvent) 652
- exclusion and synchronization 337
- executable load module
  - (286) 207
  - CODE section 210
  - DATA section 210
  - STACK section 210
- exiting a program 385
- extended memory management
  - accessing RAM disk files from terminal applications 324
  - warning 326
- ExtendedIOPHandler 553
- ExtendedIOPHandler Java class 553
  - method 553
- ExtendedIOPHandler Methods Java class
  - method
    - getStatus() 553
    - unlock(sort sState, byte byTT, byte byFF) 553
    - write(byte[] buf) 553
- extensions, naming on FAT systems 17
- extensions, naming on VFS systems 20
- external name symbols 204
- EXTERNALS option, LIB86 205

## F

- FAT (file allocation table) 161
- ferrite requirement 756
- FieldActivatedEvent Java class 597
  - methods 597
- FieldDataUpdatedEvent Java class 598
  - methods 598
- FieldDeactivatedEvent Java class 598
  - methods 598
- file
  - access 335
  - access rights 29
  - allocation space 29
  - allocation table (FAT) 161
  - attributes, changing 356
  - naming conventions 727
  - performance 37
  - pointer 336
  - pointer, shared 336, 337, 347, 349
  - services 338
  - services, general 346, 354
  - size 280
  - space allocation 29
  - type 286 207
  - use table 19
- file access priorities, terminal applications 280
- file functions
  - accessing files 30
  - copying files 33
  - creating files 29
  - deleting files 30
  - disabling file security 35
  - disabling subdirectory security 35
  - enabling file security 35
  - enabling subdirectory security 35
  - ending access to files 31
  - ensuring data integrity across PLDs 37
  - performing 29
  - protecting files 33
  - protecting subdirectories 34
  - reading a file record 35
  - renaming files 33
  - sharing files 31
  - writing a file record 35
- file in exception logs, check 375
- file names
  - for disk media 335
  - on LAN system 16
  - store application 26
  - using in terminal applications 26
- file options
  - INPUT 213
  - L86 212
- file record
  - reading 35
  - writing 35
- file security 33
- file server exception log, checking 505
- file server exception logs, retrieve 374
- files
  - access rights for 29

- files *(continued)*
  - accessing 30
  - accessing on a LAN system 27
  - application logical names 21
  - command-line 202
  - compound 348
  - compound, creating 341
  - copying 33
  - copying RAM disk from controller applications 315
  - creating 29, 348
  - creating on LAN system 29
  - creating POS 346
  - defining logical names 21
  - defining user logical names 21
  - deleting 30, 354
  - dictionary of 4690 OS 727
  - direct 12
  - distributed, in terminal applications, accessing 24
  - enable/disable security 35
  - ending access to 31
  - error recovery, disk 162
  - handling on system 8
  - keyed 12
  - keyed, creating 338
  - library 201
  - listing files on disk 278
  - listing using cross-reference map 204
  - listing using library module map 204
  - managing 11
  - naming 16
  - naming on FAT systems 17
  - naming on VFS systems 20
  - non-POS, creating 338
  - OBJ 212
  - object 207
  - open 349, 351
  - POS non-keyed, creating 346
  - printing 221
  - protecting 33
  - RAM disk 314
  - random 11
  - read a record 351
  - renaming 33, 355
  - selecting file types 11
  - sequential 14
  - sharing 31
  - space allocation for 29
  - system logical names 21
  - using logical names 20
  - write a record 352
  - write with hold 352
- filetype for overlays (OVRs) 214
- filing a report using Input Sequence Table utility 198
- FISDATA 138
- flat panel displays 759
- flush buffers (KeyedFile class) 495
- folding algorithm 183
- font change programming example 109, 124
- font specification 108, 123
- formatting data for check printing, model 1 or 2 printer 102
- free space, disk 303, 369

- freeing storage 386
- front or top loaded documents 109
- FullScreenAttributes Java class 600
  - methods 600
    - getDakAreaVideoParms() 601
    - getErrorAreaVideoParms() 601
    - getNumberOfPositions() 601
    - getPromptAreaVideoParms() 601
    - getQuoteSubstitute() 602
    - getTabLeftFCode() 601
    - getTabRightFCode() 601
    - getVideoParms(int) 601
- FullScreenFieldListener Java class 621
  - methods 621
- function code
  - defined 66
  - information 68
- FunctionCode Java class 639
  - methods 639
    - asString() 641
    - belongsToFullScreenState() 641
    - clearInputFieldData(boolean, int) 641
    - getColumn() 641
    - getCurrentFieldVideoAttributes() 641
    - getDataLengthMaximum() 641
    - getDataLengthMinimum() 641
    - getDataValueMaximum() 642
    - getDataValueMinimum() 642
    - getEmptyFieldVideoAttributes() 642
    - getHighRange() 642
    - getId() 642
    - getInputField() 642
    - getLowRange() 642
    - getNonEmptyFieldVideoAttributes() 642
    - getRow() 643
    - getStateID() 643
    - getTabOrder() 643
    - getTabOrderOfNext() 643
    - getTabOrderOfPrevious() 643
    - hasDataLengthSet() 643
    - hasDataValueSet() 643
    - isAlphanumeric() 643
    - isAutoTab() 644
    - isClearKey() 644
    - isDataAllowed() 644
    - isDataPrecedesFunctionCode() 644
    - isDataRequired() 644
    - isFullScreenDataEntry() 644
    - isKeyedLabelMayPrecede() 644
    - isKeyRange() 644
    - isLeftJustified() 644
    - isManagersKeyRequired() 645
    - isMotorKey() 645
    - isUpperCase() 645
- FunctionCodeList Java class 625
  - methods 625
    - clearInputFields(boolean) 627
    - getCurrentFunctionCode() 626
    - getFirstFunctionCode() 625
    - getFunctionCodeWithId(int) 626
    - getFunctionCodeWithTabOrder(int) 625



## FunctionCodeList Java class *(continued)*

### methods *(continued)*

- getLowestTabOrder() 626
- getNextFunctionCode(FunctionCode) 626
- getPreviousFunctionCode(FunctionCode) 626
- getStateId() 626
- getTabOrderOfCurrent() 626

### functions

- ADX\_CFILES 357
- ADXEXIT 321
- ADXFILLES 317
- ADXPSTAR 281
- ADXSTART 280
- ASCII to EBCDIC, converting 303, 370
- deleting 335
- disable controller programmable power 301
- disconnect the active user 308
- displaying application status 302
- displaying background application status 302
- EBCDIC to ASCII, converting 303, 370
- enable controller programmable power 301
- ERR 155
- ERRF% 155
- ERRL 155
- ERRN 155
- getting configured controllers on the network 303
- getting controller ID for a terminal 303
- getting disk free space 303
- HEX to ASCII, converting 388
- power off a local machine (controller or terminal) 301
- power off a remote controller 300
- power off a remote terminal 300
- PRsxCRT 287
- PRsxlNT 289
- PRsxWRC 290
- PRsxWRT 289
- query preservation flag 370
- reset preservation flag 370
- restart a background application 301
- set date and time 373
- set preservation flag 370
- set terminal sleep mode inactive timeout 306, 372
- sign off the active user 308
- stop application with message 302
- storage retention 162
- terminal dump preservation 304
- wait for keyboard and mouse activity 308
- wait for keyboard and mouse inactivity 308

## **G**

- general file services 346, 354
- generating tone 146
- get all active controllers on network 305, 371, 503
- get application status information 437
- get controller application status information, Java 504
  - determine controller ID for specified terminal 518
  - determine controller LAN status 517
  - get acting controller type 517
  - get assigned console 517
  - get display type 518

- get controller application status information, Java *(continued)*
  - get LAN connection type 519
  - get master controller ID 519
  - get monetary format 519
  - get number of digits after decimal 519
  - get number of printer lines per page 519
  - get number of terminals configured 519
  - get printer associated with console 520
  - get store number 519
  - get the date 518
  - get time format 520
- get controller machine model and type 372
- get keyboard type, Java 538
- get length of key (KeyedFile class) 496
- get loop ID for a specific terminal 504
- get loop message 304, 371, 504
- get loop status 304, 371, 505
- get record size (KeyedFile class) 496
- get terminal application status information, Java 527
  - determine if check flipper is present 536
  - determine if loop is in backup 537
  - determine if MICR is present 537
  - determine if printer is 4610 printer 537
  - determine if single-byte character set is used 537
  - determine if storage retention is enabled 537
  - determine if terminal is master 538
  - determine if terminal is online 537
  - determine printer family 537
  - determine terminal power status 538
  - get application environment 534
  - get date format 534
  - get default application name 534
  - get EC level of 4690 micro code 534
  - get monetary format 534
  - get number of digits after decimal 535
  - get NVRAM size 535
  - get partner terminal address 535
  - get store controller ID 535
  - get store number 535
  - get terminal number 535
  - get terminal type 535
  - get time format 536
  - get type of system display 535
  - get video adapter type 536
- GetAllAttributes 687
- GetAllAttributesCurrentUser 688
- GetAttribute 691
- GetAttributes 690
- GetAuthorizationLevel 699
- GetCurrentUserID 685
- GetExpires 706
- GetExpireWarning 708
- GetGroupNumber 697
- GetIDs 704
- GETLONG 116
- GetMinChange 709
- GetMinPasswordLength 707
- GetModelCount 705
- GetModelIDs 705
- GetMultipleCharacter 710
- GetRecordCount 704

- getting
  - application status 368
  - application status information 296
  - configured controllers 370
  - configured controllers on the network 303, 503
  - controller application status information 504
  - controller ID 370
  - controller ID for a specified terminal 303, 503, 505
  - controller ID for a terminal 303, 503, 505
  - disk free space 303, 369
  - file pointer 356
  - machine model 504
  - machine type 504
  - query application 527
  - storage 385
  - terminal application status information 527
- GetUserAttributes 692
- GetUserNumber 698
- Globals Java class 646
  - method 646
    - getGlobalFunctionCodes() 646
- group ID 33
- GROUP parameters 210
- guidance lights 39
  - setting on shopper display 45
- guidelines for assembly language applications 389
- guidelines for writing non-4680 BASIC programs 333

## H

- handlers and monitors, DeviceManager 539
- handling display output 548
- hardware errors
  - store controller 156
  - terminal 156
- hashing algorithm 183
- heap 385, 386
- heap data 279
- HEX to ASCII, converting 388
- HOLD option 37, 345
- home block 191
- home sensors, printer 114, 125
- HQ command (hold queue) 224

## I

- I/O device errors 165
- I/O options
  - \$M 205
  - \$O 205
  - \$X 205
- I/O processor
  - accessing 72
  - allowing operator input 74
  - characteristics of 64
  - determining status of 74
  - disallowing operator input 74
  - loading the input sequence tables 72
  - operator input 72
  - programming tips for 64
  - receiving data 73

- I/O processor (*continued*)
  - receiving data from 72
  - system input devices 65
  - waiting for data from 73
- I/O redirection to Java 539
- IBM VisualAge C/C++
  - 4690 specific behavior 676
  - compiling 666
  - creating DLLs 668
  - description 665
  - development platform 665
  - differences 674
  - differences between Windows NT and 4690 OS 674
  - environment file processing 669
  - files provided for development 665
  - JNI-specific information 673
  - limitations 674
  - linking your applications 667
  - messages 668
  - multi-byte argument support 667
  - optional link libraries 667
  - overrides for specific machine IDs or programs 670
  - platform restrictions 674
  - restrictions 674
  - run-time environment variables with 4690 668
  - running your application 672
  - support DLLs 666
  - terminal differences 676
  - unsupported features 677
  - using 665
  - wildcard expansion 667
- IBMKeylock, JavaPOS logical device 590
- IBMToneIndicator, JavaPOS logical device 590
- ID
  - authorization record 282
  - controller 370
  - controller for a specified terminal 303, 503, 505
  - controller for a terminal 303, 503, 505
  - getting controller ID for a terminal 303, 503, 505
  - group 33
  - password 282
  - user 33
- IF END# 154
- in-memory files, RAM disk 314
- indexed library 207
- information for states of input sequence table 67
- initialize PRS driver 361
- INPUT file options 213
- input file type, LIB86 201
- input file, creating 207
- input option 214
- input sequence table
  - adding a table 199
  - bar code label format 70
  - changing a table 198
  - CLEAR key 67
  - common information 66
  - copying a table 198
  - defining input sequence 66
  - displaying a table 198
  - erasing a table 198

- input sequence table *(continued)*
  - filing a report 198
  - function code definition 66
  - function code information 68
  - how to name 199
  - information for each state 67
  - input sequence definition 66
  - LAN considerations 197
  - modulo check table 70
  - modulo-check algorithm 71
  - motor function code definition 66
  - notes on using 199
  - OCR label format 69
  - on a LAN system 197
  - printing a table 198
  - renaming a table 198
  - running the utility 197
  - state definition 66
  - UPC/EAN modulo-check algorithm 71
  - user-defined modulo-check algorithm 70
  - utility options 198
- input state table utility
  - description of 197
  - tables maintained by utility 197
- InputFieldListener Java class 621
  - methods 621
    - fieldActivated(FieldActivatedEvent) 622
    - fieldDataUpdated(FieldDataUpdatedEvent) 621
    - fieldDeactivated(FieldDeactivatedEvent) 622
- InputSequenceClearedEvent Java class 650
  - method 650
    - isDoubleClear() 650
    - sequenceCleared(InputSequenceClearedEvent) 650
- InputSequenceClearedListener Java class 649
  - method 649
    - sequenceCleared(InputSequenceClearedEvent) 649
- InputSequenceFormatter Java class 647
  - methods 647
    - addInputSequenceClearedListener (InputSequenceClearedListener) 647
    - clearCurrentInputField() 647
    - getCurrentInputField() 647
    - getNumberOfBuckets() 647
    - isDataAvailable() 647
    - removeInputSequenceClearedListener (InputSequenceClearedListener) 648
- Intel 8086/80286 object module format 207
- interacting with an instance of a DeviceManager 543, 544, 545, 546
- interactive applications, definition 277
- interface
  - C language, 16-bit 334
  - C language, 32-bit 334
  - COBOL 334
- Interface DeviceManagerInterfaceTSS 545
- Interface DMDeviceRegistrationCompleteListener 545
- interface for system authorization 282
- Interface JavaInvocationHandler 545
- invoking LINK86 208
- IOPHandler Java class 550
  - methods 550
    - close() 550
    - configureBuffers(short, int) 550
    - getLastStateRead() 551

IOHandler Java class *(continued)*

- methods *(continued)*
  - isDeviceInPriorityMode() 551
  - isDeviceLocked() 551
  - isDevicePurged() 551
  - isOPDataAvailable() 551
  - loadFormatTable(byte[]) tableName 551
  - loadInputStateTable(byte[]) tableName 551
  - loadModuloTable(byte[] tableName) 551
  - lockDevice(boolean purge) 552
  - open() 552
  - read(byte[] buf) 552
  - setKeyboardData(String keyboardFileName, short index) 552
  - unlock() 552
  - unlock(short state, boolean priority) 553
  - unlock(short state) 552

IOHandler, extended 553

IOHandlerException Java class 558

- constructors 558
  - IOHandlerException() 558
  - IOHandlerException(String msg) 558
- methods 558
  - getMessage() 558
  - getReturnCode() 558
  - setAnposKbBufferOverrunRC() 559
  - setAnposKBFailureRC() 559
  - setAnposKbOfflineRC() 559
  - setBufferOverflowRC() 559
  - setDefaultRC() 559
  - setDriverLockedRC() 559
  - setInputStateTableNotLoadedRC() 559
  - setIOPOpenErrorRC() 559
  - setKBBufferOverrunRC() 559
  - setKBFailureRC() 559
  - setKBOfflineRC() 559
  - setKBOrDisplayOfflineRC() 559
  - setMatrixKbbufferOverrunRC() 560
  - setMatrixKbFailureRC() 559
  - setMatrixKbOfflineRC() 559
  - setNoResourcesRC() 560
  - setOCRBufferOverrunRC() 560
  - setOCRFailureRC() 560
  - setOCROfflineRC() 560
  - setScannerBufferOverrunRC() 560
  - setScannerFailureRC() 560
  - setScannerOfflineRC() 560
  - setStateNotValidRC() 560
  - setTableNotFoundRC() 560
  - setWriteBufferNotValidRC() 560

IOInitStatus Java class 604

- method 604
  - getStatus 604

IOInputQueue Java class 611

- methods 611
  - addLabelInputListener(LabelInputListener) 611
  - addPOSKeyListener(POSKeyListener) 612
  - addPOSKeyListenerEx(POSKeyListenerEx) 612
  - addQueueStatusListener (QueueStatusChangeListener) 612
  - getQueue() 612
  - isLocked() 612
  - isPurged() 612

- IOPInputQueue Java class *(continued)*
  - methods *(continued)*
    - postChar(char, int) 613
    - postChar(char) 612
    - postCommand(int, int) 613
    - postCommand(int) 613
    - postString(String, int) 614
    - postString(String) 613
    - removeLabelInputListener(LabelInputListener) 614
    - removePOSKeyListener(POSKeyListener) 614
    - removePOSKeyListener(POSKeyListenerEx) 614
    - removeQueueStatusListener (QueueStatusChangeListener) 614
- IOPInputQueue public data 619
- IOPReadyEvent Java class 603
- IOPReadyListener Java class 602
  - methods 602
    - iopReady(IOPReadyEvent) 602
    - iopStatus(IOPInitStatus) 602
- IOPShutdownListener Java class 608
  - method 608
    - jiopShutdown() 608
- IPL
  - command processor 239
  - disable terminal 307
  - enable terminal 307
  - function 307
- IsAvailable 685

## J

- Japan Electronics and Information Technology Industries Association statement 754
- Japanese Electrical Appliance and Material Safety Law statement 756
- Japanese power line harmonics compliance statement 756
- Japanese VCCI Council Class A statement 754
- Java 1
  - JavaTOF boot archives 261
- Java 2
  - class and parameters 265
  - JavaTOF boot archives 261
  - procedure for using JavaTOF 262
  - terminal classpath setup 263
  - using JavaTOF with 261
- Java 6 457
- Java application
  - ApplicationEvent methods 527
    - getApplication 527
    - getEventType 527
  - ApplicationManager methods 522
    - addApplicationListener 523
    - Application getFocusedApplication 523
    - Application getParentApplication 523
    - Application startApplication 523
    - ApplicationManager getApplicationManager 523
    - deregisterApplicationStartup 523
    - Enumeration getApplicationStartups 523
    - Enumeration getRunningApplication 523
    - isMultiAppEnabled 522
    - isParent 522
    - registerApplicationStartup 523
    - removeApplicationListener 523
    - stopApplication 523

- Java application *(continued)*
  - ApplicationManager methods *(continued)*
    - switchTo 523
  - ApplicationStartup methods 525
    - decrementInstance 525
    - getAppArgs 526
    - getClassname 526
    - getDetails 525
    - getDisplayName 525
    - getInstances 525
    - getMaxInstances 525
    - getStdOut 526
    - getSystemArgs 525
    - incrementInstance 525
    - isGUIStartable 525
    - isGUIStoppable 525
  - ApplicationStartupListener methods 526
    - applicationEvent 526
  - class ANDisplayHandler 548
    - methods 549
  - class ANDisplayHandlerException 557
    - constructors 557
    - methods 558
  - class CashReceiptMonitor 554
    - method 554
  - class ControllerApplicationServices 502
    - methods 503
  - class ControllerStatusData 513
    - fields 514
    - methods 517
  - class DeviceManager 540
    - handlers and monitors 539
    - methods 541
  - class ExceptionEvent 652
    - constructors 652
    - method 652
  - class ExceptionListener 652
    - method 652
  - class ExtendedIOPHandler 553
    - method 553
  - class FieldActivatedEvent 597
  - class FieldActivatedEvent methods 597
    - getActivatedFunctionCodeId 597
    - getCursorPosition() 597
    - getStateId 597
  - class FieldDataUpdatedEvent 598
  - class FieldDeactivatedEvent 598
    - FieldDeactivatedEvent methods 598
      - getDeactivatedFunctionCodeId 598
      - getStateId 598
  - class FieldUpdatedEvent methods 598
    - getChangeSource() 600
    - getCursorPosition() 599
    - getDoNotInitializeScreenData() 600
    - getFunctionCode() 599
    - getFunctionCodeId() 599
    - getStateId() 599
    - getUpdatedData() 599
    - isFullScreenField() 599
  - Class FlexosException 478
    - constructors 478



Java application (*continued*)

- Class FlexosException (*continued*)
  - methods 478
- class FullScreenAttributes 600
  - methods 600
- class FunctionCode 639
  - methods 639
- class FunctionCodeList 625
  - methods 625
- class Globals 646
  - method 646
- class InputSequenceClearedEvent 650
  - method 650
- class InputSequenceClearedListener 649
  - method 649
- class InputSequenceFormatter 647
  - methods 647
- Class InvalidParameterException 478
  - constructors 479
- class IOPHandler 550
  - methods 550
- class IOPHandlerException 558
  - constructors 558
  - methods 558
- class IOPInitStatus 604
  - method 604
- class IOPInputQueue 611
  - methods 611
- class IOPReadyEvent 603
- class IOPReadyListener 602
  - methods 602
- class IOPShutdownListener 608
  - method 608
- Class JavaInvocationResult 546
- class JIOProcessor 593
  - constructor 593
  - methods 593
- class KeyedFile 488
  - constructors 490
  - fields 489
  - methods 495
- class LabelInputEvent 627
  - methods 627
- class MSRHandler 554
  - method 554
- class MSRHandlerException 560
  - constructors 560
  - methods 561
- class multiapp.Application 524
- Class multiapp.Application methods 524
  - ApplicationStartup getApplicationStartup 524
  - getExitValue 524
  - getStatus 524
  - isParent 524
  - String getDisplayName 524
- class multiapp.ApplicationEvent 526
- class multiapp.ApplicationListener 526
- class multiapp.ApplicationManager 522
- class multiapp.ApplicationStartup 525
- class multiapp.MultiAppStartup 520
- class POSFile 479

Java application (*continued*)

- fields 480
- Class POSFile methods 485
  - closeFull() 485
  - closePartial() 485
  - finalize() 485
  - getFilePointer() 485
  - length() 485
  - lock(int, int, int, int, int) 485
  - read(byte[], int, int, int, int) 486
  - seek(int) 487
  - skipBytes(int) 487
  - unlock(int, int, int) 487
  - write(byte[], int, int, short, int) 487
  - writeAppend(byte[], int) 488
- class POSKeyListener
  - method 622
- class POSPipeInputStream 496
  - constructors 497
  - methods 498
- class POSPipeOutputStream 499
  - constructors 499
  - methods 500
- class POSPrinterHandler 555
  - method 555
- class POSPrinterHandlerException 562
  - constructors 562
  - methods 562
- class PromptChangeEvent 607
  - methods 607
- class PromptChangeListener 605
  - method 605
- class QueueLockedException 628
- class QueueStatusChangeEvent 616
  - method 616
- class QueueStatusChangeListener 615
  - method 615
- class State 635
  - methods 635
- class StateChangeEvent 634
  - methods 634
- class StateForbidsInputException 628
- class StateTableProcessor 629
  - methods 629
- class SystemBusyEvent 610
  - method 610
- class SystemBusyListener 609
  - method 609
- class SystemMonitor
  - constructor 651
  - methods 651
- class TerminalApplicationServices 527
  - methods 527
- class TerminalKeyboardLights 547
- class TerminalStatusData 530
  - fields 531
  - methods 534
- class VideoParms 616
  - method 616
- class FullScreenFieldListener
  - methods 621

- Java application *(continued)*
  - configuring as a primary application 457
  - configuring as a secondary application 457
  - constructors 482
    - POSFile(File, String, int, int, short, byte) 482
    - POSFile(File, String, int, int) 482
    - POSFile(File, String, int) 482
    - POSFile(String, String, int, int, short, byte) 484
    - POSFile(String, String, int, int) 484
    - POSFile(String, String, int) 483
  - DBCS support 657
  - designing applications 455
  - Interface DeviceManagerInterface 543
  - Interface DeviceManagerInterfaceMultiJVM 544
  - Interface DeviceManagerInterfaceTSS 545
  - Interface DMDeviceRegistrationCompleteListener 545
  - interface FullScreenFieldListener 621
  - interface InputFieldListener 621
    - methods 621
  - Interface JavaInvocationHandler 545
  - interface LabelInputListener 624
    - methods 624
  - interface POSKeyListener 622
  - interface StateChangeListener 633
    - method 633
  - public data IOInputQueue 619
- Java application start 311
- Java application stop 312
- Java GUI functions 587
- Java I/O processor functions 586
- Java I/O redirection 539
- Java IO processor classes 593
- Java IO processor configuration 590
- Java IO processor packages 592
- Java methods
  - disable controller programmable power 503
  - disconnect active user 511
  - enable controller programmable power 508
  - getting configured controllers on the network 503
  - getting controller ID for a terminal 503, 505
  - power off a local machine (controller or terminal) 506, 528
  - power off a remote controller 506
  - power off a remote terminal 507
  - query application 505
  - query terminal dump preservation flag 505
  - reset terminal dump preservation flag 508
  - set terminal dump preservation flag 508
  - set terminal sleep mode inactive timeout 508
  - sign off active user 510
  - wait for keyboard and mouse inactivity 510
  - wait for keyboard or mouse activity 510
- Java screen switch 311
- JavaTOF
  - boot archives 261
  - creating a BAT file 266
  - Java 2 class and parameters 265
  - Java 2 terminal classpath setup 263
  - multi-app properties file 264
  - procedural enhancements 266
  - procedure with Java 2 262
  - properties files 264

- JavaTOF (*continued*)
  - terminal RAM disk preload (Java 2) 265
  - with Java 2 261
  - with multi-app 263
- JavaTOF property file editor 256
- JIOp code example 591
- JIOp definitions and concepts 587
- JIOp trace file
  - capturing 654
- JIOProcessor Java class 593
  - constructor 593
    - JIOProcessor(IOPReadyListener ior[]) 593
  - methods 593
    - addIOPReadyListener(IOPReadyListener) 594
    - addIOPShutdownListener (IOPShutdownListener) 594
    - addPromptChangeListener (PromptChangeListener) 594
    - addSystemBusyListener (SystemBusyListener) 594
    - getInputQueue() 595
    - getInputSequenceFormatter() 595
    - getIOPProcessor() 595
    - getPrompt() 595
    - getProperties() 595
    - getTableProcessor() 595
    - isReady() 595
    - isSystemBusy() 595
    - removeIOPReadyListener (IOPReadyListener) 595
    - removeIOPShutdownListener (IOPShutdownListener) 596
    - removePromptChangeListener (PromptChangeListener) 596
    - removeSystemBusyListener (SystemBusyListener) 596
- journal buffering 111

## K

- keyboard and mouse, wait for activity 308, 375
- keyboard and mouse, wait for inactivity 308, 375
- keyed file utility
  - creating an empty keyed file 179
  - creating direct files with 178
  - creating keyed files with 178
  - from a host site 177
  - keyed file internal processes 185
  - rebuilding a keyed file 179
  - reporting keyed file statistics 179
  - using 177
    - using in a batch file 179
- keyed file utility, services 338
- keyed files
  - create 338
  - creating direct file from 179
  - creating empty file using batch method 181
  - delete keyed record 346
  - description of 12
  - internal processes of 185
  - key folding 185
  - key transformation 185
  - open file 342
  - performance 38
  - randomizing 185
  - read record 343, 344
  - write keyed record 345
- KeyedFile class, Java 488

KeyedFile class, Java *(continued)*

- constructors 490
  - KeyedFile(File, String, int, byte, byte, int, int, int, int, int) 490
  - KeyedFile(File, String, int, int, int, int, int, int) 492
  - KeyedFile(file, string, int) 490
  - KeyedFile(string, string, int, byte, byte, int, int, int, int, int) 493
  - KeyedFile(string, string, int, int, int, int, int, int) 494
  - KeyedFile(string, string, int) 492
- fields 489
  - COMPOUND\_FILE 489
  - DISTRIBUTE\_ON\_CLOSE 489
  - DISTRIBUTE\_ON\_UPDATE 489
  - EXCLUSIVE\_ACCESS 489
  - HOLD 489
  - LOCK 489
  - MIRRORED\_FILE 489
  - NO\_HOLD 489
  - NO\_LOCK 490
  - NO\_UNLOCK 490
  - SHARED\_READ\_ACCESS 490
  - SHARED\_READ\_WRITE\_ACCESS 490
  - UNLOCK 490
- methods 495
  - closeFull() 495
  - closePartial() 495
  - delete(byte[]) 495
  - finalize() 495
  - getKeyLength() 496
  - getRecordSize() 496
  - read(byte[]), int) 496
  - write(byte[], int, int) 496
- Korean communications statement 754

## L

- L86 file options 212
- label format
  - bar code 70
  - magnetic ticket 69
  - OCR 69
- LabelInputEvent Java class 627
  - methods 627
    - getData() 627
    - getFunctionCode() 627
    - isScannerInput() 627
- LabelInputListener Java class
  - methods 624
    - invalidKeyedLabel(string) 624
    - labelInput(LabelInputEvent) 624
- LabelInputListener Java interface 624
- LAN considerations for input sequence table utility 197
- LAN system
  - accessing files on 27
  - building logical names table 23
  - changing logical names table 23
  - creating files on 29
  - displaying logical names table 23
  - distribution exception log 161
  - filenames on 16
  - input sequence table on 197
  - logical file names on 22

- LAN system (*continued*)
  - logical file names table 22
- language translators 207
- large memory model, defined 278
- left home command 114
- LIB86
  - appending an existing library 203
  - creating a library file 202
  - error messages 731
  - input filetype 201
  - LIB86 204
  - listing files 204
  - using 201
- LIB86 options
  - abbreviations for 201
  - accessing files in other directories 205
  - command-line options 201
  - DELETE 203
  - EXTERNALS 205
  - MAP 205
  - MODULES 205
  - NOALPHA 205
  - PUBLICS 205
  - SEGMENTS 205
- libraries, searching 215
- library files
  - creating 204
  - deleting modules from 203
  - description of 201
  - public names in 204
  - renaming 203
  - replacing modules in 203
  - runtime with terminal services 287
  - selecting modules from 204
  - using the indexed library 207
- library utility 207
- LIBSYMS option 211
- LIN file options 211
- LINES option 212
- link path variables 215
- LINK86
  - command line 217
  - command options 209
  - command-file options 210
  - error codes 738
  - error messages 734
  - how to invoke 208
  - input option 214
  - L86 file options 212
  - LIN file options 211
  - linking with shared run-time libraries 209
  - MAP file options 212
  - output option 214
  - SYM file options 211
- linkage editor 207
- linker utility 207
- listing
  - directories of terminal applications 315
  - files on disk 278
  - files using ADXDIR 322
  - files using cross-reference map 204

- listing (*continued*)
  - files using library module map 204
  - library files, LIB86 204
  - listing directories RAM disk file from 315
- listing terminal hard disk files 443
- listing terminal RAM disk files 443
- loading
  - a module 210
  - a module header 210
- local area network (LAN)
  - application read restrictions 167
  - erasing spool file on 168
  - forcing spool file erasure 169
  - spool files on 167
  - user application considerations 167
  - using D drive for spool file 169
  - using TCLOSE on 167
  - using the audible alarm function on 160
  - WRITE MATRIX spooling 168
- local file, using 341, 348, 357
- local symbols 211
- LOCALS options 211
- Lock 692
- LOCK 353
- locking a record 353
- logging
  - application events 312
  - distribution errors 28
  - distribution exception 161
  - errors 155, 383
  - system errors 155
- logging an application event 442
- logging an error 506, 528
- logical file names
  - application 21
  - CHAIN 26
  - defining user 21
  - on LAN system 22
  - system 21
  - user 21
- logical names
  - defining 21
  - using 20
- logical names table
  - building 23
  - changing 23
  - displaying 23
- logical node name 27
- logo printing, 4610 printer 126
- logo printing, model 1 or 2 printer 103
- logo printing, model 3 or 4/4A printer 114
- long file names
  - name server database 28
  - overview 28
  - restrictions 28
  - using 28
- loop
  - get ID for a specific terminal 504
  - get message 304, 371, 504
  - get status 304, 371, 505
- LQ command (print spooler) 224

## M

- magnetic ink character recognition
  - data format 118
  - determining if installed 119
  - errors, understanding 119
  - parsing routines 120
  - reader, using 119
  - reading from 119
  - support 118
- magnetic ink character recognition support, model 3 and 4/4A printers 118
- magnetic stripe reader
  - accessing 89
  - common data characteristics 89
  - determining status of 90
  - disallowing data 90
  - dual-track characteristics 85
  - example of code for dual-track 92
  - example of code for JIS-II MSR 95
  - example of code for single-track 91
  - example of code for three-track 93
  - JIS-II MSR characteristics 85
  - JUCC MSR characteristics 85
  - preparing to receive data from 89
  - receiving data from 89
  - single-track characteristics 85
  - three-track characteristics 85
- magnetic ticket label format 69
- managing
  - files 11
  - files, disk 335
  - pipes 336
- managing devices, Java class for 540, 543, 544, 545, 546, 547, 548
- managing input/output processes, Java class for 550
- managing input/output processing exceptions, Java class for 558, 560, 562
- manual or automatic document insertion 110
- Map (MAP) file 207, 209
- MAP file options 212
- MAP option, LIB86 205
- maps, module 204
- master exception log, checking 505
- master exception logs, retrieve 374
- MAXIMUM parameter for LINK86 210, 211
- media, disk 335
- medium memory model, defined 278
- memory
  - allocating 385
  - allocating using ADDMEM 392
  - parameter block 386
  - when code size exceeds available 278
- memory management 325
- memory models
  - big 278
  - large 278
  - medium 278
- mercury-added statement 759
- message list, audible alarm 159
- message size, pipes 289
- message types used with pipes 288
- message, get loop 304, 371, 504
- methods
  - disable controller programmable power 503



- methods *(continued)*
  - disconnect active user 511
  - enable controller programmable power 508
  - getting configured controllers on the network 503
  - getting controller ID for a terminal 503, 505
  - power off a local machine (controller or terminal) 506, 528
  - power off a remote controller 506
  - power off a remote terminal 507
  - query application 505
  - query terminal dump preservation flag 505
  - reset terminal dump preservation flag 508
  - set terminal dump preservation flag 508
  - set terminal sleep mode inactive timeout 508
  - sign off active user 510
  - wait for keyboard and mouse inactivity 510
  - wait for keyboard or mouse activity 510
- mirrored file, using 341, 348, 357
- miscellaneous functions errors, CPI 453
- model 2 printer driver 99
- model 2 printer, characteristics 99
- model 3 printer driver 106
- model 3 printer, characteristics 106
- model 3/4/4A MICR support 118
- model 4/4A printer driver 106
- model 4/4A printer, characteristics 106
- Modify incomplete dump sequence 295
- module map 204
- module, root 218
- modules
  - loading 210
  - loading a header 210
  - object 201
- MODULES option, LIB86 205
- modulo check algorithm, IBM 71
- modulo check table 70
- modulo-check algorithm 71
- modulo-check algorithm, UPC/EAN 71
- modulo-check algorithm, user-defined 70
- monitoring the cash receipt 554
- monitors and handlers, DeviceManager 539
- motor function code 66
- MSR, programming tips for 84
- MSRHandler 554
- MSRHandler Java class 554
  - method 554
    - close() 554
    - getStatus() 555
    - isDeviceLocked() 555
    - isMSRDataAvailable() 555
    - lock(boolean bLock) 554
    - open() 554
    - read(byte[] buf) 554
- MSRHandlerException Java class 560
  - constructors 560
    - MSRHandlerException() 560
    - MSRHandlerException(String msg) 561
  - methods 561
    - getMessage() 561
    - getReturnCode() 561
    - setBufferTooSmallRC() 561
    - setDataNotCompleteRC() 561

- MSRHandlerException Java class *(continued)*
  - methods *(continued)*
    - setDevAlreadyOpenedRC() 561
    - setDeviceNotAttachedRC() 561
    - setDeviceOfflineRC() 561
    - setDevNotAttachedRC() 561
    - setKBOOrDualTrkNotAttachedRC() 561
    - setUnableToReadRC() 561
- multi-app
  - Java 2 terminal classpath setup 263
  - properties files 264
  - using JavaTOF 263
- multiapp.Application Java class 524
- multiapp.ApplicationEvent Java class 526
- multiapp.ApplicationListener Java class 526
- multiapp.ApplicationManager Java class 522
- multiapp.ApplicationStartup Java class 525
- multiapp.MultiAppStartup Java class 520

## N

- naming
  - conventions for 4690 OS files 727
  - extensions, on FAT systems 17
  - files and subdirectories 16
  - subdirectories, files on FAT systems 17
  - subdirectories, files, VFS systems 20
  - tables for input sequence table 199
- nested overlays 217
- NOALPHA option 205
- NOALPHA option, LIB86 205
- node name use in accessing files 27
- nodename, using to access files 27
- NOLIBSYMS option 211
- NOLINES option 212
- NOLOCALS option 209, 211
- non-4680 BASIC programs, writing 333
- non-POS file or pipe, creating 338
- nondestructive read 337, 351
- NOSHARED options 213
- NOSYMS option 211
- notices 751
  - battery recycling 758
  - cable ferrites 756
  - electronic emissions 752
  - electrostatic discharge (ESD) 756
  - end of life disposal 757
  - Toshiba 751

## O

- OBJ files 212
- object file 207
- object file error codes 738
- object module format, Intel 8086/80286 207
- OCR
  - label format 69
  - label format, magnetic ticket 69
- ON ASYNC ERROR CALL 154
- ON ERROR 153
- OPEN 342

- open a distributed keyed file, Java class to create and 490
- open a local keyed file, Java class to create and 492
- open an existing keyed file Java class 490, 492
- open distributed keyed file Java class, create 493
- open file or pipe 349
- open keyed file 342
- open local keyed file Java class, create 494
- open serial statement, opening port 141
- operating system
  - C interface 333
  - COBOL interface 333
  - environment 3, 277
  - protected-mode 277
- operating system, protected-mode 277
- operator authorization 364
- operator input, I/O processor 72
- operator interactive applications 277
- operator interface, system 4
- operator record
  - adding to authorization file 283
  - changing password 283
  - deleting from authorization file 283
- option
  - \$ option 214
  - \$C (command) option 214
  - \$D (debug) option 214
  - \$L (library) option 214
  - \$M (map) option 214
  - \$N (line number) option 214
  - \$O (object) option 215
  - \$S (symbol) option 215
  - DBINFO option 215
- optional emulator report 392
- options for LIB86, command-line 201
- output option 214
- overlay
  - (OVR) file 207
  - chaining to 313
  - command line syntax 217
  - nesting 217
  - OVR (overlay filetype) 214
- overlay command line syntax 217
- OVR (overlays) file 207

## P

- package com.ibm.OS4690.jiop 593
- package com.ibm.OS4690.jiop.util 650
- packages, Java IO processor 592
- parsing routines for model 3 or 4/4A printer driver 120
- partial close 343, 351
- password
  - authorization of 364
  - changing in operator record 283
  - definition of 365
  - encryption 285, 366
  - ID 282
- password rules, enhanced security 679
- password, enhanced
  - APIs 684
  - application programming interfaces 684

- password, enhanced (*continued*)
  - authorization attributes 680
  - Cancel 712
  - ChangePassword 687
  - Commit 711
  - Copy 693
  - Create 701
  - Delete 702
  - error codes 681
  - GetAllAttributes 687
  - GetAllAttributesCurrentUser 688
  - GetAttribute 691
  - GetAttributes 690
  - GetAuthorizationLevel 699
  - GetCurrentUserID 685
  - GetExpires 706
  - GetExpireWarning 708
  - GetGroupNumber 697
  - GetIDs 704
  - GetMinChange 709
  - GetMinPasswordLength 707
  - GetModelCount 705
  - GetModelIDs 705
  - GetMultipleCharacter 710
  - GetRecordCount 704
  - GetUserAttributes 692
  - GetUserNumber 698
  - IsAvailable 685
  - Lock 692
  - password rules 679
  - Rename 702
  - sessions 683
  - SetAttribute 695
  - SetAttributes 694
  - SetAuthorizationLevel 700
  - SetExpires 700, 706
  - SetExpireWarning 708
  - SetGroupNumber 697
  - SetMinChange 709
  - SetMinPasswordLength 707
  - SetMultipleCharacter 711
  - SetPassword 703
  - SetUserAttributes 696
  - SetUserNumber 698
  - StartSession 689
  - StartSessionCurrentUser 690
  - user ID rules 679
  - user-defined attributes 681
  - using 679
  - Validate 686
  - ValidateID 686
- perchlorate 759
- performance
  - design considerations for files 37
  - file 37
  - keyed files 38
- performing file functions, creating files 29
- permanent background applications 277
- pipe
  - access to 337
  - conditional write 290

- pipe (*continued*)
  - creating 348
  - definition of 286
  - deleting files or pipes 354
  - management 336
  - message size 289
  - non-POS, creating 338
  - PRS, conditional write to 363
  - PRS, creating 361
  - PRS, read 361
  - PRS, write to 362
  - read a record 351
  - routing services 287, 361
  - types of messages used with 288
  - using 286
- pipe routing services errors, CPI 452
- pipe services 496
  - close file stream (POSPipeInputStream) 498
  - close output pipe (POSPipeOutputStream) 500
  - close output stream (POSPipeOutputStream) 500
  - create input file stream (POSPipeInputStream) 497
  - create input file, remote pipe (POSPipeInputStream) 497
  - open output file stream to local pipe (POSPipeOutputStream) 500
  - open output file stream to remote pipe 500
  - read bytes (POSPipeInputStream) 498
  - read next byte (POSPipeInputStream) 498
  - read record from file (POSPipeInputStream) 498
  - read specified byte length (POSPipeInputStream) 499
  - release system resources (POSPipeOutputStream) 500
  - skip data bytes from input stream (POSPipeInputStream) 499
  - write a record (POSPipeInputStream) 498
  - write byte (POSPipeOutputStream) 501
  - write byte conditionally to remote pipe (POSPipeOutputStream) 502
  - write bytes specified by buffer length (POSPipeOutputStream) 500, 501
  - write conditionally to remote pipe from offset (POSPipeOutputStream) 501
- pipes
  - deleting 336
  - open 349, 351
  - read a record 351
  - write a record 352
- PJ command (print spooler) 223
- PLD data integrity 37
- PLD protection, when writing 37
- PLD recovery
  - enable/disable IPL 306
  - recovery 161
  - store controller 161
  - terminal 161
- polynomial hashing algorithm
  - example application using 191
  - using 189
- POS files
  - open a distributed POS file 482
  - open a local POS file 482
  - open a local POS file with specified name 484
- POS keyboard type 538
- POSFile class 479
  - append a record to file 488
  - class POSFile methods 485
  - close the file stream 485
  - create a distributed POS file 482

POSFile class *(continued)*  
     create a distributed POS file with specified name 484  
     create a local POS file 482  
     create a local POS file with specified name 484  
     fields 480  
         COMPOUND\_FILE 480  
         DISTRIBUTE\_ON\_CLOSE 480  
         DISTRIBUTE\_ON\_UPDATE 480  
         EXCLUSIVE\_ACCESS 480  
         FLUSH 480  
         FROM\_CURRENT\_FILE\_POINTER 480  
         FROM\_END\_OF\_FILE 480  
         FROM\_START\_OF\_FILE 480  
         LOCK\_EXCLUSIVE 480  
         LOCK\_SHARED 481  
         LOCK\_WRITE 481  
         MIRRORED\_FILE 481  
         NO\_FLUSH 481  
         NO\_WAIT\_ON\_LOCK 481  
         READ\_FROM\_CACHE 481  
         READ\_FROM\_DISK 481  
         SHARED\_READ\_ACCESS 481  
         SHARED\_READ\_WRITE\_ACCESS 481  
         WAIT\_ON\_LOCK 481  
     finalize 485  
     flush the buffer 485  
     get file pointer 485  
     get length of file 485  
     lock a record 485  
     open a distributed POS file with specified name 484  
     open an existing POS file 482  
     open an existing POS file with specified name 483  
     read record from file 486  
     seek file pointer 487  
     skip bytes 487  
     unlock a record 487  
     write a record 487  
 positioning the print head 113  
 POSKeyListener Java class 622  
     methods 622  
         dataKeyPress(char) 622  
         functionKeyPress(int) 622  
 POSPipeInputStream class, Java 496  
     constructors 497  
         POSPipeInputStream(int, char) 497  
         POSPipeInputStream(string, int) 497  
     methods 498  
         available() 498  
         close() 498  
         finalize() 498  
         read() 498  
         read(byte[], int, int) 499  
         read(byte[]) 498  
         skip(long) 499  
 POSPipeOutputStream class, Java 499  
     constructors 499, 500  
         close() 500  
         finalize() 500  
         POSPipeOutputStream(char, string) 500  
         POSPipeOutputStream(string) 500  
         write(byte[], int, int) 501

- POSPipeOutputStream class, Java *(continued)*
  - constructors *(continued)*
    - write(byte[]) 500
    - write(int) 501
    - writeConditional(byte[], int, int) 501
    - writeConditional(byte[]) 501
    - writeConditional(int) 502
- POSPrinterHandler 555
- POSPrinterHandler Java class 555
  - method 555
    - close(int nStation, boolean bTClose) 555
    - getStatus(int nStation) 556
    - isMICRDataAvailable() 557
    - open(int nStation) 555
    - read(byte[] buf) 556
    - setStatus(int nStation, int nStatus) 557
- POSPrinterHandlerException Java class 562
  - constructors 562
    - POSPrinterHandlerException() 562
    - POSPrinterHandlerException(String msg) 562
  - methods 562
    - getMessage() 562
    - getReturnCode() 562
    - set4689CRNoPaperRC() 564
    - set4689SJNoPaperRC() 564
    - setBufferOverflowRC() 562
    - setCmdRejectRC() 563
    - setCoverOpenButPrintLineOkRC() 564
    - setCoverOpenErrorRC() 563
    - setDefaultRC() 562
    - setDeviceFailureRC() 563
    - setDeviceNotAttachedRC() 563
    - setDeviceOfflineRC() 562
    - setDIDocMissingRC() 563
    - setDIDocNotAllowedRC() 563
    - setDIStationErrorRC() 563
    - setDocModeNotValidRC() 563
    - setFintFileNotDownLoadedRC() 563
    - setFlashEpromRC() 564
    - setInvalidOperationRC() 562
    - setInvalidOptionRC() 564
    - setJrnBufferOverflowRC() 564
    - setJrnPaperErrRC() 563
    - setKeyPressedErrorRC() 563
    - setMicrReadErrorRC() 564
    - setNoMicrRC() 563
    - setNoPowerErrorRC() 564
    - setNoResourcesRC() 564
    - setOverlayCharsNotValidRC() 564
    - setPrinterNoStorageRC() 564
    - setPrinterOrDriverBufferRC() 564
    - setReceivedIllegalDataRC() 562
    - setReceivedIllegalParamRC() 564
    - setTimedOutOnReadRC() 563
    - setWriteLogoNotSupportedRC() 563
- POSTLINK error messages 732
- POSTLINK utility
  - description of 218
  - ERRORLEVEL test 219
  - invoking the utility 218

- power
  - disable controller programmable power 301, 503
  - enable controller programmable power 301, 508
  - power off a local machine (controller or terminal) 301, 506, 528
  - power off a remote controller 300, 506
  - power off a remote terminal 300, 507
- predefined subdirectory 727
- preparing to receive data from the magnetic stripe reader 89
- primary applications 277
- print spooler 221
- print spooler commands
  - AJ 224
  - AQ 224
  - CJ 224, 225
  - HJ 224
  - HQ 224
  - LJ 224
  - PJ 223
  - RJ 225
  - TJ 223, 225
- print spooler error codes 225
- printer
  - accessing 100
  - accessing 4610 122
  - accessing 4689 128
  - accessing model 3 107
  - accessing model 4/4A 107
  - application functions 100
  - application functions, 4610 122
  - application functions, model 3 or 4/4A 107
  - character sets, model 1 or 2 printer 103
  - characteristics of 4610 120
  - characteristics of 4689 127
  - characteristics of model 2 99
  - characteristics of model 3 106
  - characteristics of model 4/4A 106
  - compatibility 106, 121
  - controlling document insert station, 4610 124
  - controlling document insert station, model 1 or 2 100
  - controlling document insert station, model 3 or 4/4A 109
  - cutter control characters 114, 125
  - determination, 4610 124
  - determination, 4689 129
  - determination, model 3 or 4/4A 113
  - determining the status, model 1 or 2 101
  - determining when printing is complete, 4610 127
  - determining when printing is complete, 4689 129
  - determining when printing is complete, model 1 or 2 104
  - determining when printing is complete, model 3 or 4/4A 115
  - document eject command 112, 124
  - document options 111
  - document station character line lengths 113, 124
  - emphasized printing 108
  - example of code for, model 1 or 2 104
  - font change example 109, 124
  - font specification 108, 123
  - front or top loaded documents 109
  - home sensors 114, 125
  - journal buffering 111
  - left home command 114
  - logo printing, 4610 126



- printer (*continued*)
  - logo printing, model 1 or 2 103
  - logo printing, model 3 or 4/4A 114
  - manual or automatic document insertion 110
  - MICR support for model 3 and model 4/4A 119
    - common MICR errors 120
    - data format 115
    - parsing routines 120
  - performance considerations, model 1 or 2 104
  - performance considerations, model 3 or 4/4A 115
  - positioning the print head 113
  - preparing a line to print, model 1 or 2 100
  - preparing a line to print, model 3 or 4/4A 107
  - preventing unnecessary reprints 113
  - printing a line of text, 4610 123
  - printing a line of text, model 1 or 2 100
  - printing a line of text, model 3 or 4/4A 107
  - printing example 108
  - reading from 138
  - receipt paper cutter control characters 114, 125
  - receipt paper cutter example 114, 125
  - receipt paper cutter, 4610 125
  - receipt paper cutter, model 3 or 4/4A 114
  - reinserting documents 112
  - removing and replacing a document 111
  - reversible document station motor 113
  - top or front loaded documents 109
- printer determination, 4610 124
- printer determination, 4689 129
- printer determination, model 3 or 4/4A 113
- printer driver
  - 4610 printer 120
  - 4689 printer 127
  - compatibility with Model 1 and 2 106
  - compatibility with Model 3 and 4/4A 121
  - model 2 99
  - model 3 106
    - MICR support 115
  - model 4/4A 106
    - MICR support 115
  - programming tips for 99, 106, 120, 127
- printer, 4689 native mode support 130
- printing
  - checks, 4610 printer 125
  - checks, model 1 or 2 printer 101
  - checks, model 3 or 4/4A printer 114
  - files using print spooler 221
  - logo, 4610 126
  - logo, model 1 or 2 printer 103
  - logo, model 3 or 4/4A 114
  - table using Input Sequence Table utility 198
- printing programming example 108
- priorities for searching 216
- priority 383
- problem determination aids 8
- problem determination for check printing, model 1 or 2 printer 103
- procedural enhancements, JavaTOF 266
- PROCESS table 385
- processing special commands 670
- processing, suspended 386
- programmable power 439

- programmable power (*continued*)
  - disable controller programmable power 301, 503
  - enable controller programmable power 301, 508
  - power off a local machine (controller or terminal) 301, 506, 528
  - power off a remote controller 300, 506
  - power off a remote terminal 300, 507
- programming, application 138
- programs
  - ending 384
  - writing non-4680 333
- PromptChangeEvent Java class 607
  - methods 607
    - getColumn() 607
    - getPrompt() 607
    - getRow() 607
- PromptChangeListener Java class 605
  - method 605
    - promptChanged(PromptChangeEvent) 605
- properties files
  - JavaTOF 264
  - multi-app 264
- protecting
  - files 33
  - subdirectories 34
  - system operator authorization file 282
- PRS driver, initializing 361
- PRS pipe, conditional write to 363
- PRS pipe, write to 362
- PRS, read from a PRS pipe 361
- PRScRT function 287
- PRScINT function 289
- PRScWRC function 290
- PRScWRT function 289
- public name symbols 204
- public names in library files 204
- public symbols 201
- PUBLICS option, LIB86 205
- PUTLONG 117
- PWIN\$ parameter for password encryption 285
- PWOUT\$ parameter for password encryption 285
- Python 713

## Q

- query application 505
- query terminal application, Java 527
- query terminal dump preservation flag 370, 505
- QueueLockedException Java class 628
- QueueStatusChangeEvent Java class 616
  - method 616
    - isLocked() 616
- QueueStatusChangeListener Java class 615
  - method 615
    - queueStatusChange(QueueStatusChangeEvent) 615

## R

- RAM disk files
  - accessing from controller applications 315
  - accessing from terminal applications 315
  - copying from terminal applications 315

- RAM disk files *(continued)*
  - in-memory 314
  - listing directories of terminal applications 315
- RAM disk preload for Java 2 265
- RAM disks
  - configuration considerations 314
  - copying files from controller applications 315
- random file access 336
- random files 11
- RCP command 242
- reactivating configured file server 170
- read 39
  - attributes from the video display 53
  - characters from a 2x20 display 42
  - characters from the shopper display 45
  - characters from the video display 52
  - commands
    - GETLONG 116
    - PUTLONG 117
  - file record 35, 351
  - keyed record 344
  - Master/File Server activation and deactivation 378
  - nondestructive 337, 351
  - pipe record 351
  - PRS pipe 361
- read bytes specified by buffer length (POSPIPEInputStream) 498
- READ MATRIX 35
- read next data byte (POSPIPEInputStream) 498
- read record from file (KeyedFile class) 496
- read specified byte length (POSPIPEInputStream) 499
- reading totals retention driver data in direct mode 148
- receive paper cutter
  - control characters 114, 125
  - example 114, 125
- receiving data from the I/O processor 72
- receiving data from the serial I/O communications driver 142
- record, unlocking a 353
- recovering from errors 153
- recovery from PLD
  - enable/disable IPL 306
  - on store controller 161
  - on terminal 161
- redirected devices, configured to Java 540
- redirection errors and exceptions 557
- redirection, Java I/O 539
- reinserting documents for printing 112
- release system resources (POSPIPEOutputStream) 500
- remote STC, run in all terminals 307, 373
- remote STC, run in one terminal 307, 373
- removing and replacing a document 111
- Rename 702
- renaming
  - files 33, 355
  - library files 203
  - table using input sequence table utility 198
- REPLACE option 203
- replacing library modules 203
- reporting keyed file statistics 179
- reports
  - "a" errors 393
  - "b" startup data 393

- reports (*continued*)
  - “c” interrupt trace 393
  - “d” OUT, OUTS, IN, INS trace 393
  - optional emulator 392
  - selecting using EOPTIONS 393
- requesting an application service 293
- requesting an application service (ControllerApplicationServices) 502
- reset terminal dump preservation flag 370, 508
- restrictions
  - on application reads 167
- restrictions for assembly language applications 389
- restrictions for writing non-4680 BASIC programs 333
- restrictions when accessing long file name support 28
- RESUME 154
- RESUME label 155
- RESUME RETRY 155
- retention, storage 162
- retrieve file server exception logs 374
- retrieve master exception logs 374
- return codes 334
- reversible document station motor 113
- root module 218
- routing services, pipes 287, 361
- run remote STC in all terminals 307, 373
- run remote STC in one terminal 307, 373
- run-time exception handling 672
- running input sequence table on LAN system 197
- runtime library 287

## S

- safety information xxv
- scale driver
  - accessing 139
  - characteristics of 139
  - example of code for 139
  - programming tips for 139
  - receiving data 139
- scanner
  - ADXNScan API 76
  - ADXSCAN API 76
  - allow applications direct access to 76
- scanner driver
  - example of code for 79
- SEARCH option 212
- search priorities 216
- searching libraries 215
- secondary applications 277
- section, DATA 210
- security 277, 364
- security for files and subdirectories, enable/disable 35
- seek file pointer 356
- segment name symbols 204
- SEGMENT parameters 210
- SEGMENTS option, LIB86 205
- selecting file types 11
- selecting modules from library file 204
- semaphore 337
- sequential access 336
- sequential files
  - description of 14

- sequential files *(continued)*
  - example of 15
  - open 14
  - using file pointers 336
  - write record to 14
- serial I/O communications driver
  - accessing 141
  - characteristics of 141
  - determining serial I/O port status 143
  - preparing to receive data from 142
  - preparing to transmit data to 144
  - programming tips for 141
  - receiving data from 142, 143
  - sending a break to 144
  - simultaneous receive and transmit 144
  - transmitting data to 144
  - waiting for data from 143
- services
  - file 338
  - files and pipes 346
  - pipe routing 287
  - terminal 287
- set date and time 373
- set terminal dump preservation flag 370, 508
- set terminal sleep mode inactive timeout function 306, 372, 508
- SetAttribute 695
- SetAttributes 694
- SetAuthorizationLevel 700
- SetExpires 700, 706
- SetExpireWarning 708
- SetGroupNumber 697
- SetMinChange 709
- SetMinPasswordLength 707
- SetMultipleCharacter 711
- SetPassword 703
- setting
  - the error level value using ADXEXIT 321
- SetUserAttributes 696
- SetUserNumber 698
- severity codes 157
- shareable runtime libraries (SRTL) 213
- shared file pointer 336, 347, 349
- shared file pointer, pipes 337
- SHARED options 213
- sharing files 31
- shopper display
  - accessing 45
  - characteristics of 44
  - clearing 45
  - determining guidance light status 45
  - display character set 45
  - example of code 45
  - programming tips for 44
  - reading from 45
  - setting guidance lights on 45
  - writing characters to 45
- sign off the active user 308, 376
- signon screen, changing 4
- single-track MSR
  - accessing 89
  - characteristics 85

- single-track MSR (*continued*)
  - common data characteristics 89
  - determining status of 90
  - disallowing data 90
  - example of code for 91
  - preparing to receive data from 89
  - receiving data from 89
  - restrictions 88
  - waiting for received data 89
- size
  - application 278
  - code 278
  - data 279
  - file 280
- skip data bytes from input stream (POSPipeInputStream) 499
- sleep mode, terminal 306, 372
- software interrupts 394, 395
- space for file allocation 29
- special commands processing 670
- special system facilities 8
- specialized services 364
- specification, font 108, 123
- spool files
  - erasing 168
  - using D drive for 169
- spool files on LAN system 167
- spooling
  - spooling 221
  - WRITE MATRIX 168
- SRTL (shareable runtime libraries) 213
- stack data 279
- STACK section 210
- staged IPL utility
  - application interface 242
  - applications only 241
  - apply software maintenance 242
  - capabilities 241
  - disable terminal IPL 246
  - enable terminal IPL 246
  - error recovery 243
  - how to use 243
  - incompatible software levels 241
  - messages 246
  - RCP command 242
  - requirements 241
  - tcc network 241
  - terminal storage 246
- staged IPL utility messages 246
- start Java application 311
- starting a background application 280, 382
- starting a background application with a priority 281
- StartSession 689
- StartSessionCurrentUser 690
- State Java class 635
  - methods 635
    - clearFullScreenInputFields(boolean) 636
    - getActiveFieldFunctionCode() 636
    - getAutoEOFCount() 636
    - getAutoEOFFunctionCode() 636
    - getID() 636
    - getLowestTabOrder() 636

- State Java class (*continued*)
  - methods (*continued*)
    - getNumberOfFunctionCodes() 636
    - getPrompt() 636
    - getValidFullScreenFunctionCodes() 637
    - getValidFunctionCodes() 637
    - isAutoEOF() 637
    - isDataDisplayedPerFCode() 637
    - isDataDisplayedWithEditing() 637
    - isDataDisplayedWithOutEditing() 637
    - isFullScreenState() 637
    - isKeyboardAllowed() 637
    - isScannerAllowed() 637
    - isSequenceStart() 638
  - state, definition 66
- StateChangeEvent Java class 634
  - methods 634
    - getState() 634
    - getStateId() 634
    - isChangeToCurrent() 634
- StateChangeListener Java class
  - method 633
    - stateChanged(StateChangeEvent) 633
- StateChangeListener Java interface 633
- StateForbidsInputException Java class 628
- statement
  - CREATE 26
  - IF END# 154
  - ON ASYNC ERROR CALL 154
  - ON ERROR 153
  - OPEN 26
  - READ MATRIX 35
  - RENAME 26
  - RESUME 154
  - RESUME label 155
  - RESUME RETRY 155
  - SIZE 26
- StateTableProcessor Java class
  - methods 629
- StateTableProcessor methods
  - activateField(FunctionCode, boolean) 630
  - addFullScreenFieldListener(FullScreenFieldListener, int) 630
  - addFullScreenFieldListener(FullScreenFieldListener) 630
  - addInputFieldListener(InputFieldListener) 630
  - addStateChangeListener(StateChangeListener) 630
  - getActiveFunctionCode() 630
  - getCurrentState() 631
  - getCurrentStateId() 631
  - getFullScreenAttributes() 631
  - getGlobals() 631
  - getNumberOfStates() 631
  - getState(int) 631
  - isFullScreenTableLoaded() 631
  - removeFullScreenScreenFieldListener(FullScreenFieldListener, int) 632
  - removeFullScreenScreenFieldListener(FullScreenFieldListener) 631
  - removeInputFieldListener(InputFieldListener) 632
  - removeStateChangeListener (StateChangeListener) 632
- static data area 279
- status
  - application 367, 368
  - of background application 278

- status, get loop 304, 371, 505
- STC, run remotely in all terminals 307, 373
- STC, run remotely in one terminal 307, 373
- stop Java application 312
- stopping an application with message 302, 369
- storage
  - freeing 386
  - getting 385
  - retention 162
  - retention, terminal, enable 295, 508, 529
- store application file names 26
- store closed query, BASIC application 320
- store controller
  - backup 9
  - errors 156
  - PLD recovery 161
  - system messages at the 156
- store controller application services 317
- store controller services 293
- store controller-unique application services 321
- string, FISDATA 138
- subdirectory
  - enable/disable security 35
  - naming on FAT systems 17
  - naming, VFS systems 20
  - predefined 727
  - protecting 34
- suspended processing 386
- switch terminal screen 311
- SYM (Symbol File) options 211
- SYM (Symbol Table) file 207
- Symbol File (SYM) options 211
- Symbol Table (SYM) file 207
- symbols
  - external name 204
  - public 201
  - public name 204
  - segment name 204
  - unresolved 207
- synchronization and exclusion 337
- system
  - authorization 282
  - capabilities 4
  - errors 156
  - messages 156
  - messages at the store controller 156
- system authorization, interface to 282
- system capabilities
  - backup 9
  - communications 7
  - concurrent operations 4
  - file handling 8
  - operator interface 4
  - problem determination aids 8
  - special facilities 8
- system date, get 431
- system error log 155
- system function mode 311
- system input devices
  - description of 65
- system logical file names 21



- system messages
  - at the store controller 156
  - at the terminal 157
  - audible alarm 159
  - explanation 156
  - severity codes 157
- system operator authorization file
  - definition of 282
  - protection on 282
  - subdirectory where found 282
- system storage dump 295, 367, 503, 527
- SystemBusyEvent Java class 610
  - method 610
    - getSystemBusyClearKey() 610
- SystemBusyListener Java class 609
  - method 609
    - systemBusy(SystemBusyEvent) 609
- SystemMonitor Java class 651
  - constructor 651
    - SystemMonitor() 651
  - methods 651
    - addExceptionListener(ExceptionListener) 651
    - getInstance() 651
    - removeExceptionListener(ExceptionListener) 651
    - uncaughtException(Thread, Throwable) 651

## T

- table, erasing using the input sequence table utility 198
- table, logical names
  - building 22
  - changing 22
  - displaying 22
- Taiwanese battery recycling statement 758
- TCLOSE 167
- temporary background applications 277
- terminal
  - disable IPL 307
  - enable IPL 307
  - errors 156
  - getting controller ID for 303, 503, 505
  - power off 162
    - power off for Mod1 162
    - power off for Mod2 162
  - power on 162
    - power on for Mod1 162
    - power on for Mod2 162
  - set date and time 373
  - set terminal sleep mode inactive timeout function 306, 372, 508
  - system messages at the 157
- terminal application
  - accessing RAM disk files from 315
  - chaining 314
  - distributed files in 24
  - WRITE MATRIX record size for 170
- terminal application query 527
- terminal application services 293, 322
  - start Java application 311
  - stopping Java application 312
- terminal application status information 527
- terminal C applications 333

- terminal classpath setup 263
- terminal default data size 279
- terminal dump preservation function 304
- terminal offline function (JavaTOF) 249
  - dependency checking 250
  - dependency checking limitations 250
  - DependencyChecking class 250
  - JavaTOF classes 250
  - overview, JavaTOF 258
  - property file editor 256
  - requirements for applications using the JavaTOF solution 249
  - resource creation 250
  - using dependency checking 250
- terminal PLD recovery 161
- terminal RAM disk preload (Java 2) 265
- terminal screen switch 311
- Terminal Services program 157
- terminal storage retention
  - at the 4683-xx1 162
  - at the 4683-xx2 162
  - disable function 296
  - disable method 503, 527
  - enable function 295
  - enable method 508, 529
- terminal, run remote STC 307, 373
- TerminalApplicationServices Java class 527
  - methods 527
    - disableIPL() 527
    - disableStorageRetention() 527
    - dumpSystemStorage() 527
    - getTerminalStatus() 527
    - isTerminal() 527
    - logError(char, int, int, int, byte[]) 528
    - powerOffLocalMachine(String) 528
    - setIPL() 528
    - setStorageRetention() 529
- TerminalKeyboardLights Java class 547
  - methods
    - isMsgPendLightOn() 547
    - isOfflineLightOn() 547
    - isWaitLightOn() 547
    - readMsgPendMessage() 548
    - readOfflineMessage() 548
    - readWaitMessage() 547
- terminals, run remote STC in al 373
- terminals, run remote STC in all 307
- TerminalStatusData Java class 530
  - fields 531
    - ANDISPLAY 531
    - ANDISPLAY2 531
    - ANDISPLAY3 531
    - COMMA\_CONVENTION 532
    - CONTROLLER\_TERMINAL 532
    - D\_M\_Y\_FORMAT 532
    - DAY\_MONTH\_YEAR\_FORMAT 532
    - HMS\_TIME\_FORMAT 532
    - HOURL\_MINUTE\_SECONDS\_FORMAT 532
    - M\_D\_Y\_FORMAT 532
    - MONTH\_DAY\_YEAR\_FORMAT 532
    - NVRAM\_SIZE\_16K 532
    - NVRAM\_SIZE\_1K 532

TerminalStatusData Java class *(continued)*

fields *(continued)*

PERIOD\_CONVENTION 532  
TERMINAL 533  
TERMINAL\_4683 533  
TERMINAL\_4684 533  
TERMINAL\_4693 533  
TERMINAL\_4694 533  
TERMINAL\_4800 533  
TWO\_SIGNIFICANT\_DIGITS 533  
VDISPLAY 533  
VDISPLAY2 533  
VGA\_VIDEO\_DISPLAY 533  
VIDEO\_DISPLAY\_FOR\_4683 533  
ZERO\_SIGNIFICANT\_DIGITS 534

methods 534

getApplicationEnvironment() 534  
getDateFormat() 534  
getDefaultApplicationName() 534  
getEcLevelof4610MicroCode() 534  
getMonetaryFormat() 534  
getNumberOfDigitsAfterDecimal() 535  
getNVRAMSize() 535  
getPartnerTerminalAddress() 535  
getStoreControllerId() 535  
getStoreNumber() 535  
getSystemDisplayType() 535  
getTerminalNumber() 535  
getTerminalType() 535  
getTimeFormat() 536  
getVideoAdapterType() 536  
isFliperPresentOn4610() 536  
isLoopInBackup() 537  
isMicrPresentOn4610() 537  
isPrinterA4610() 537  
isPrinterATi1OrTi2 537  
isSBCSInUseOn4610() 537  
isStorageRetentionEnabled() 537  
isTerminalOnLine() 537  
isTerminalPoweredOn() 538  
isTerminalTheMaster() 538

test, ERRORLEVEL 216

three-track MSR

accessing 89  
characteristics 85  
common data characteristics 89  
data formats 86  
determining status of 90  
disallowing data 90  
error reporting 86  
example of code for 93, 95  
preparing to receive data from 89  
receiving data from 89  
restrictions 88  
waiting for received data 89

time frame indicators, Disk Surface Analysis Utility 239

time-slice method 280

TIMER 386

TJ command (print spooler) 223

tone driver

accessing 146

- tone driver (*continued*)
  - characteristics of 146
  - example of code for 147
  - generating tone 146
  - programming tips for 146
- top or front loaded documents 109
- total disk space 278
- totals retention driver
  - accessing 148
  - accessing in direct mode 148
  - characteristics of 147
  - example of code for 149
  - programming tips for 147
  - reading data in direct mode 148
  - reading data, sequential mode 149
  - specifying address in sequential mode 149
  - tone driver 147
  - writing data in direct mode 148
  - writing data, sequential mode 149
- touch screen
  - programming tips for 39
- TQ command (print spooler) 225
- trace dump file
  - capturing 656
- trace JIOP file
  - capturing 654
- trademarks 760
- translators, language 207
- types of background applications 277

## U

- unlocking a record 353
- unresolved symbols 207
- UPC/EAN modulo-check algorithm 71
- use factor 201
- user defined modulo-check algorithm 70
- user ID 33
- user ID rules, enhanced passwords 679
- user logical file names 21
- using
  - application services 293
  - D drive for spool file 169
  - file names in store controllers 26
  - file names in terminal applications 26
  - logical names 20
  - node names to access files 27
- utilities
  - keyed file utility 177
  - LIB86 201
  - POSTLINK 218
  - print spooler utility 221
  - staged IPL 241
- utility options for input sequence table 198

## V

- Validate 686
- ValidateID 686
- variables 277
- variables, link path 215

- verifying definitions 185
- VFS name server database 28
- VFS translation 28
- video display
  - accessing 50
  - attributes 39
    - Feature A video driver attribute 48
    - reading 53
    - VGA video driver Feature A attribute emulation 49
    - VGA video driver VGA attribute 49
    - writing 52
  - changing the video display format 50
  - character location wrapping 51
  - characteristics of Feature A video driver 47
  - characteristics of VGA video driver 48
  - clearing 50
  - current character attribute 51
  - current character location 51
  - determining display information 52
  - display character set 52
  - example of code for 53
  - example of command stacking 58
  - Feature A video driver 47
  - Feature A video driver video display formats 47
  - programming tips for 47
  - reading attributes from the video display 53
  - reading characters from the video display 52
  - reading from the video display 52
  - restrictions using command stacking 58
  - running a 2x20 display application 53
  - using Feature A video driver command stacking 57
  - VGA video driver 47
  - VGA video driver video display formats 48
  - writing attributes without changing characters 52
  - writing characters without changing attributes 52
  - writing to 51
- VideoParms Java class 616
  - method 616
    - getAttributes() 618
    - getBottomBorderChar() 619
    - getBottomLeftBorderChar() 619
    - getBottomRightBorderChar() 619
    - getColumn() 618
    - getFlags() 617
    - getLeftBorderChar() 618
    - getRightBorderChar() 618
    - getRow() 617
    - getTopBorderChar() 618
    - getTopLeftBorderChar() 618
    - getTopRightBorderChar() 618
    - getWidth() 618
    - hasBorder() 617
    - is2x20() 617
    - isCentered() 617
- virtual file system translation 28
- VisualAge C/C++
  - 4690 specific behavior 676
  - compiling 666
  - creating DLLs 668
  - description 665
  - development platform 665

## VisualAge C/C++ (continued)

- differences 674
- differences between Windows NT and 4690 OS 674
- environment file processing 669
- files provided for development 665
- JNI-specific information 673
- limitations 674
- linking your applications 667
- messages 668
- multi-byte argument support 667
- optional link libraries 667
- overrides for specific machine IDs or programs 670
- platform restrictions 674
- restrictions 674
- run-time environment variables with 4690 668
- running your application 672
- support DLLs 666
- terminal differences 676
- unsupported features 677
- using 665
- wildcard expansion 667

## W

- wait for keyboard and mouse activity 308, 375
- wait for keyboard and mouse inactivity 308, 375
- waiting for
  - data from I/O processor 73
  - data from the serial I/O communications driver 143
  - for event completion 387
- warning, check printing 103
- when code size exceeds available memory 278
- windowing 277
- windows 277
- write 39
  - attributes to video display 52
  - characters only to video display 52
  - characters to a 2x20 display 40
  - characters to the shopper display 45
  - characters with current character attribute to video display 51
  - file records 35, 352
  - non-4680 BASIC programs 333
  - pipe records 352
  - totals retention driver data in direct mode 148
- write a record (KeyedFile class) 496
- write byte (POSPipeOutputStream) 501
- write byte conditionally to remote pipe (POSPipeOutputStream) 502
- write bytes specified by buffer length (POSPipeOutputStream) 500, 501
- write conditionally to remote pipe (POSPipeOutputStream) 501
- write conditionally to remote pipe from offset (POSPipeOutputStream) 501
- write keyed record 345
- WRITE MATRIX spooling 168
- write to PRS pipe 362
- write verify function for hard disk drive, disabling 38
- write with hold 37, 353

## X

- XOR rotation hashing algorithm 187





Product Number: 5639-P70

G362-0574-02

