

z/VM
7.4

CP Exit Customization



Note:

Before you use this information and the product it supports, read the information in [“Notices” on page 295](#).

This edition applies to version 7, release 4 of IBM® z/VM® (product number 5741-A09) and to all subsequent releases and modifications until otherwise indicated in new editions.

Last updated: 2024-09-18

© **Copyright International Business Machines Corporation 1995, 2024.**

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Figures.....	ix
Tables.....	xi
About This Document.....	xiii
Intended Audience.....	xiii
Syntax, Message, and Response Conventions.....	xiii
Where to Find More Information.....	xvi
Links to Other Documents and Websites.....	xvi
How to provide feedback to IBM.....	xvii
Summary of Changes for z/VM: CP Exit Customization.....	xix
SC24-6269-74, z/VM 7.4 (September 2024).....	xix
SC24-6269-73, z/VM 7.3 (December 2023).....	xix
SC24-6269-73, z/VM 7.3 (September 2022).....	xix
SC24-6269-02, z/VM 7.2 (March 2021).....	xix
SC24-6269-01, z/VM 7.2 (September 2020).....	xix
Chapter 1. Introduction.....	1
Benefits of Using CP Exits.....	1
Understanding the System Execution Space and CP Customization Methods.....	1
Storage Addresses.....	2
Customizing CP Using the SYSGEN Method.....	2
Customizing CP Using the CP Exit Method.....	3
Loading Files into the SXS Dynamic Area.....	3
Unloading Files from the SXS Dynamic Area.....	3
Creating, Controlling, and Redefining CP Commands and Diagnose Codes.....	4
Creating, Controlling, and Calling CP Exits.....	4
Creating, Controlling, and Using Local CP Message Repositories.....	4
Chapter 2. Migrating Your Local Modifications.....	5
Techniques for Isolating Source Modifications.....	5
Exploit an IBM-Defined CP Exit Point.....	5
Create Your Own Exit Point.....	6
Front-end an Existing Diagnose Code.....	6
Front-end an Existing CP Command.....	7
Eliminating Source Modifications.....	7
Command Table Updates.....	7
Diagnose Code Table Updates.....	8
Defining Linkage Attributes for New Modules.....	8
Selecting a Name for New Modules.....	8
Changes to the System Common Area.....	8
Changes to the CP Message Repository.....	8
Dynamically Loaded Routines.....	9
Chapter 3. Creating a Dynamically Loaded Routine.....	11
Input Data.....	12
Output Data.....	12

Addressing Mode.....	13
Attributes.....	13
Using CP Services.....	16
Defining System Linkage to Your Routine.....	16
Discussion of the HCPPROLG Macro.....	16
Discussion of the HCPENTER Macro.....	17
Discussion of the HCPEXIT Macro.....	18
Discussion of the HCPCALL Macro.....	18
HCPCALL TYPE=DIRECT.....	18
HCPCALL TYPE=INDIRECT.....	20
Usage Note for HCPCALL.....	21
Discussion of the HCPLCALL Macro.....	21
Discussion of the HCPLINTR Macro.....	22
Discussion of the HCPLEXIT Macro.....	22
Discussion of the HCPCONSL Macro.....	22
Some HCPCONSL Examples.....	28
Using Other Dynamically Loaded Routines.....	29
How Should the Routine be Named?.....	30
How Should the Linkage Attributes of the Routine be Specified?.....	30
How are Addresses Resolved?.....	31
Can I Add My Own CP IUCV System Service?.....	31
What Does the CPXUNLOAD ASYNCHRONOUS Operand Mean?.....	31
What Does the CPXLOAD CONTROL Operand Imply?.....	32
What Happens if I Make a Programming Error?.....	32
Chapter 4. Loading Dynamically into the System Execution Space.....	33
Understanding CPXLOAD.....	33
Using the DELAY and NODELAY Operands.....	33
Using the PERMANENT and TEMPORARY Operands.....	33
Using the CONTROL and NOCONTROL Operands.....	34
Using the MP, NOMP, and NONMP Operands.....	34
Using the LET and NOLET Operands.....	34
How to Package Modules.....	35
Examples of CPXLOAD Commands.....	35
CPXLOAD Example 1: Installing a New CP Command.....	36
CPXLOAD Example 2: Installing a New Diagnose Code.....	36
CPXLOAD Example 3: Installing a New Message.....	37
Examples of CPXLOAD Directives.....	37
Chapter 5. Controlling a Dynamically Loaded Routine.....	39
Controlling a Dynamically Loaded CP Command Routine.....	39
Controlling a Dynamically Loaded Diagnose Code Routine.....	40
Controlling a Dynamically Loaded CP Exit Routine.....	40
Controlling a Dynamically Loaded Local Message Repository.....	41
IPL Parameter NOEXITS.....	41
Chapter 6. Defining and Modifying Commands and Diagnose Codes.....	43
CMDBKs, DGNBKS and You.....	43
CMDBKs.....	43
DGNBKS.....	43
Relationship between IBM Class and User Privilege Class.....	44
Coding Your Command and Diagnose Code.....	44
Coding a Command Handler.....	44
Coding a Diagnose Code Handler.....	46
Defining your Command and Diagnose Code.....	47
Defining Your Command.....	48
Defining Your Diagnose Code.....	48

Defining an Alias Command.....	49
Modifying a Command or Diagnose Code.....	50
Disabling a Command or Diagnose Code.....	50
Chapter 7. Defining and Using CP Message Repositories.....	51
Why Use Message Repositories?.....	51
Components of a Message.....	51
Creating a Message Repository File.....	52
Example of a Message Repository.....	52
Generating the Object Repository File.....	53
Loading and Unloading Message Files.....	54
Example 1.....	54
Example 2.....	54
Producing Messages from a Repository.....	55
How Does CP Choose Which Repository to Use?.....	55
Producing Messages Without Substitution.....	56
Producing Messages With Substitution.....	56
Producing Messages With Column Format.....	57
When You Cannot Use a Message Repository.....	59
Chapter 8. Unloading Dynamically from the System Execution Space.....	61
What May Be Unloaded.....	61
CPXLOAD Load ID Number.....	62
When to Use ASYNCHRONOUS and SYNCHRONOUS.....	62
Chapter 9. Understanding the CP Parser.....	63
Introduction to the Syntax Definition Macros.....	63
Understanding How to Code for the Parser Using an Example.....	64
Step 1: Develop the Syntax Railroad Track Diagram.....	64
Step 2: Assign HCPCFDEF Macro to the First Token.....	65
Step 3: Mark Alternative Path Locations.....	65
Step 4: Assign HCPSTDEF Macro to Alternative Path Locations.....	66
Step 5: Assign HCPTKDEF Macro to Each Token.....	67
Step 6: Add the Keywords for the Macros.....	67
Step 7: Develop the Storage Area Definitions.....	69
Step 8: Develop the HCPDOSYN Macro.....	70
Step 9: Write the Code to Call the Parser.....	71
Step 10: Write the Commands to Activate the Code.....	71
Using the Parser to Store Data.....	71
Dividing Your Syntax over Multiple Modules.....	74
Combining ROOT=, PLBASE=, BASES=, and SYNLIST=.....	75
Using a Post-Processor.....	75
Providing Additional Processing in Your Mainline Routine.....	76
Providing Additional Processing in Your Post-processor.....	76
Using a Single Syntax for a Command and for a Configuration File Statement.....	76
Creating Your Own Configuration File Statements.....	77
Using One Syntax for Two Purposes.....	78
Error Messages and How the Parser Handles Them.....	79
Detecting Syntax Errors.....	80
Additional Features.....	80
Chapter 10. Common and Frequent Problems.....	81
Available Diagnostic Facilities.....	81
Common Mistakes.....	83
Command Related Problems.....	83
Diagnose Code Related Problems.....	84
Message Related Problems.....	85

Exit Point Related Problems.....	85
CPXLOAD Related Problems.....	86
CPUNXLOAD Related Problems.....	87
Miscellaneous CP Customization Errors.....	87
Appendix A. IBM-Defined CP Exit Points.....	89
Summary of IBM-Defined CP Exit Points.....	90
Usage Conventions.....	93
Standard Point of Processing.....	93
Standard Entry Conditions.....	93
Standard Parameter List Contents.....	94
Standard Exit Conditions.....	94
Standard Return Codes.....	94
Additional Information about Control Entry Points.....	95
CP Exit 00E0: Startup Pre-Prompt Processing.....	97
CP Exit 00E1: Startup Post-Prompt Processing.....	101
CP Exit 0FFB: Post-Authorization Command Processing.....	105
CP Exit 1100: LOGON Command Pre-Parse Processing.....	109
CP Exit 1101: Logon Post-Parse Processing.....	112
CP Exit 1110: VMDBK Pre-Logon Processing.....	114
CP Exit 117F: Logon Final Screening.....	115
CP Exit 11C0: Logoff Initial Screening.....	117
CP Exit 11FF: Logoff Final Screening.....	118
CP Exit 1200: DIAL Command Initial Screening.....	120
CP Exit 1201: DIAL Command Final Screening.....	122
CP Exit 1210: Messaging Commands Screening.....	123
CP Exit 1230: VMRELOCATE Eligibility Checks.....	129
CP Exit 1231: VMRELOCATE Destination Restart.....	130
CP Exit 3FE8: SHUTDOWN Command Screening.....	131
CP Exit 4400: Separator Page Data Customization.....	133
CP Exit 4401: Separator Page Pre-Perforation Positioning.....	135
CP Exit 4402: Separator Page Perforation Printing or 3800 Positioning.....	137
CP Exit 4403: Separator Page Printing.....	138
CP Exit 4404: Second Separator Page Positioning.....	140
CP Exit 4405: Second Separator Page Printing.....	142
CP Exit 4406: Separator Page Post-Print Positioning.....	143
CP Exit 4407: Trailer Page Processing.....	145
CP Exit Point and Module Reference.....	147
Appendix B. Dynamic Exit Points.....	149
Point of Processing.....	149
Entry Conditions.....	149
Parameter List Contents.....	149
Exit Conditions.....	149
Return Codes.....	149
Appendix C. CPXLOAD Directives.....	151
CHANGE Directive.....	152
EXPAND Directive.....	154
INCLUDE Directive.....	156
OPTIONS Directive.....	158
Appendix D. CP Files of Interest.....	161
CP Control Blocks and Macros.....	161
CP Modules.....	163
Appendix E. CP Accounting Exit (Module HCPACU).....	165

Entry Point HCPACUON.....	166
Entry Point HCPACUOF.....	167
Appendix F. CP Exit Macros.....	169
HCPXSERV: CP Exit Services Director.....	170
Example 1.....	178
Example 2.....	179
Example 3.....	179
MDLATENT: MDLAT Entry Definition.....	180
MDLATHDR: MDLAT Header.....	186
MDLATTLR: MDLAT Trailer.....	187
Appendix G. CP Parser Macros and Routines.....	189
HCPCFDEF: Command/Config File Statement Definition Macro.....	190
HCPDOSYN: Parser Syntax Table Generator Macro.....	191
HCPSTDEF: Parser State Definition Macro.....	193
Example 1.....	194
Example 2.....	194
Example 3.....	195
HCPTKDEF: Parser Token Definition Macro.....	196
Example 1.....	204
HCPZPRPC — Parse Remainder of Command Line.....	205
HCPZPRPG — Parse Any GSDBK.....	206
Pre-Processing Routines.....	206
Post-Processing Routines.....	207
Appendix H. Understanding the CP Message Repository.....	209
Message Repository File.....	209
Commenting Your Message Repository.....	209
Creating a Control Line.....	209
Creating Message Records.....	209
Message Repository Idiosyncrasies.....	211
Additional CP Message Repository Idiosyncrasies.....	211
Appendix I. Updating the CP Load List.....	213
General Rules for Coding an Alternate MDLAT Macro.....	213
Creating an Alternate MDLAT Macro.....	214
Coding the User Module.....	215
Generating a New CP Load List.....	216
Appendix J. Samples of Dynamically Loaded Routines.....	217
What You Should Gain from the Sample CP Exit 1200 Routines.....	218
Understanding CP Exit 1200.....	218
Understanding the Sample CP Exit 1200 Routines.....	218
Sample 1.....	219
Sample 2.....	219
Why Sample 2 Is Better Than Sample 1.....	220
Using the Sample CP Exit 1200 Routines.....	220
Potential Improvements.....	221
A Sample JAMTABLE SOURCE File.....	291
Appendix K. I/O Monitor User Exit.....	293
Call Mechanics.....	293
Important Usage Notes.....	293
Serialization.....	294

Notices.....	295
Programming Interface Information.....	296
Trademarks.....	296
Terms and Conditions for Product Documentation.....	296
IBM Online Privacy Statement.....	297
 Bibliography.....	 299
Where to Get z/VM Information.....	299
z/VM Base Library.....	299
z/VM Facilities and Features.....	300
Prerequisite Products.....	302
Related Products.....	302
 Index.....	 303

Figures

1. Layout of the SXS Using the SYSGEN Method.....	2
2. Layout of the SXS Using the CP Exits Method.....	3
3. RULES member of DGN1FC TXTLIB.....	38
4. Sample Repository - SPGMES REPOS.....	52
5. Sample Assembler Code Accessing SPGMES REPOS from module HCPXXX.....	56
6. Sample Repository - JCRMES REPOS.....	56
7. Sample Assembler Code Accessing JCRMES REPOS from module HCPXXX.....	57
8. Sample Repository - JAFMES REPOS.....	58
9. Sample Assembler code accessing JAFMES REPOS.....	59
10. Module HCPACU Overview.....	165
11. HCPSAVBK Save Area.....	166
12. AONPARM Parameter List for Entry Point HCPACUON.....	167
13. AOFPARM Parameter List for Entry Point HCPACUOF.....	168
14. CP Message Record Format.....	210
15. ABCMDLAT Macro.....	214

Tables

1. Examples of Syntax Diagram Conventions.....	xiii
2. Various Linkage Decisions.....	19
3. IBM Class Definitions.....	44
4. Results if a message is found or not found in a repository.....	55
5. Summary of IBM-Defined CP Exit Points.....	90
6. CPXLOAD Directives.....	151
7. Legal and Illegal HCPXSERV Action/Parameter Combinations.....	174
8. Register Contents During HCPXSERV Execution.....	175
9. Register Contents After HCPXSERV Completion.....	176
10. Register Contents After HCPXSERV Completion Continued.....	177
11. Conversion types.....	198
12. Annotated Listing for Sample CP Exit Routine 1 for CP Exit 1200.....	223
13. Annotated Listing for Sample CP Exit Routine 2 for CP Exit 1200.....	242
14. Annotated Listing of a Local Message Repository for Sample 2 of CP Exit 1200.....	289

About This Document

This document contains information about using CP Exits support to customize an IBM® z/VM® system. CP Exits support allows you to make nondisruptive additions and deletions of dynamically loaded CP routines (non-IBM-supplied code) to add and delete CP commands, Diagnose codes, locally developed CP message repositories, and CP exit routines.

This document describes the terminology, concepts, and functions of CP Exits support. It explains how to create and manipulate dynamically loaded routines. It also includes reference information describing the IBM-defined exit points, CPXLOAD directives, and CP exit macros.

Intended Audience

This information is intended only for experienced programmers who need to make local modifications to a z/VM system.

We recommend that you do not attempt to use the CP Exits support unless you have the following skills:

- A working knowledge of IBM Assembler Language
- Some knowledge of what the source for a CP module looks like
- Some knowledge of the CP commands associated with the CP Exits support





Syntax, Message, and Response Conventions

The following topics provide information on the conventions used in syntax diagrams and in examples of messages and responses.

How to Read Syntax Diagrams

Special diagrams (often called *railroad tracks*) are used to show the syntax of external interfaces.

To read a syntax diagram, follow the path of the line. Read from left to right and top to bottom.

- The  symbol indicates the beginning of the syntax diagram.
- The  symbol, at the end of a line, indicates that the syntax diagram is continued on the next line.
- The  symbol, at the beginning of a line, indicates that the syntax diagram is continued from the previous line.
- The  symbol indicates the end of the syntax diagram.

Within the syntax diagram, items on the line are required, items below the line are optional, and items above the line are defaults. See the examples in [Table 1 on page xiii](#).


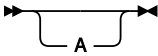
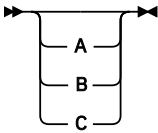
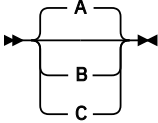
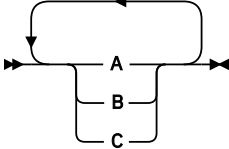

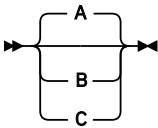
Table 1. Examples of Syntax Diagram Conventions	
Syntax Diagram Convention	Example
Keywords and Constants A keyword or constant appears in uppercase letters. In this example, you must specify the item KEYWORD as shown. In most cases, you can specify a keyword or constant in uppercase letters, lowercase letters, or any combination. However, some applications may have additional conventions for using all-uppercase or all-lowercase.	

Table 1. Examples of Syntax Diagram Conventions (continued)

Syntax Diagram Convention	Example
Abbreviations <p>Uppercase letters denote the shortest acceptable abbreviation of an item, and lowercase letters denote the part that can be omitted. If an item appears entirely in uppercase letters, it cannot be abbreviated.</p> <p>In this example, you can specify KEYWO, KEYWOR, or KEYWORD.</p>	<p>» KEYWOrd «</p>
Symbols <p>You must specify these symbols exactly as they appear in the syntax diagram.</p>	<p>* Asterisk</p> <p>: Colon</p> <p>, Comma</p> <p>= Equal Sign</p> <p>- Hyphen</p> <p>() Parentheses</p> <p>. Period</p>
Variables <p>A variable appears in highlighted lowercase, usually italics.</p> <p>In this example, <i>var_name</i> represents a variable that you must specify following KEYWORD.</p>	<p>» KEYWOrd — <i>var_name</i> «</p>
Repetitions <p>An arrow returning to the left means that the item can be repeated.</p> <p>A character within the arrow means that you must separate each repetition of the item with that character.</p> <p>A number (1) by the arrow references a syntax note at the bottom of the diagram. The syntax note tells you how many times the item can be repeated.</p> <p>Syntax notes may also be used to explain other special aspects of the syntax.</p>	<p>» repeat «</p> <p>» , repeat «</p> <p>» repeat ¹ «</p> <p>Notes: ¹ Specify <i>repeat</i> up to 5 times.</p>
Required Item or Choice <p>When an item is on the line, it is required. In this example, you must specify A.</p> <p>When two or more items are in a stack and one of them is on the line, you must specify one item. In this example, you must choose A, B, or C.</p>	<p>» A «</p> <p>» A B C «</p>

Table 1. Examples of Syntax Diagram Conventions (continued)	
Syntax Diagram Convention	Example
<p>Optional Item or Choice</p> <p>When an item is below the line, it is optional. In this example, you can choose A or nothing at all.</p> <p>When two or more items are in a stack below the line, all of them are optional. In this example, you can choose A, B, C, or nothing at all.</p>	 
<p>Defaults</p> <p>When an item is above the line, it is the default. The system will use the default unless you override it. You can override the default by specifying an option from the stack below the line.</p> <p>In this example, A is the default. You can override A by choosing B or C.</p>	
<p>Repeatable Choice</p> <p>A stack of items followed by an arrow returning to the left means that you can select more than one item or, in some cases, repeat a single item.</p> <p>In this example, you can choose any combination of A, B, or C.</p>	
<p>Syntax Fragment</p> <p>Some diagrams, because of their length, must fragment the syntax. The fragment name appears between vertical bars in the diagram. The expanded fragment appears in the diagram after a heading with the same fragment name.</p> <p>In this example, the fragment is named "A Fragment."</p>	 <p>A Fragment</p> 

Examples of Messages and Responses

Although most examples of messages and responses are shown exactly as they would appear, some content might depend on the specific situation. The following notation is used to show variable, optional, or alternative content:

xxx

Highlighted text (usually italics) indicates a variable that represents the data that will be displayed.

[]

Brackets enclose optional text that might be displayed.

{ }

Braces enclose alternative versions of text, one of which will be displayed.

|

The vertical bar separates items within brackets or braces.

...

The ellipsis indicates that the preceding item might be repeated. A vertical ellipsis indicates that the preceding line, or a variation of that line, might be repeated.

Where to Find More Information

For information about related documents, see the [“Bibliography” on page 299](#).

Links to Other Documents and Websites

The PDF version of this document contains links to other documents and websites. A link from this document to another document works only when both documents are in the same directory or database, and a link to a website works only if you have access to the Internet. A document link is to a specific edition. If a new edition of a linked document has been published since the publication of this document, the linked document might not be the latest edition.

How to provide feedback to IBM

We welcome any feedback that you have, including comments on the clarity, accuracy, or completeness of the information. See [How to send feedback to IBM](#) for additional information.

Summary of Changes for z/VM: CP Exit Customization

This information includes terminology, maintenance, and editorial changes. Technical changes or additions to the text and illustrations for the current edition are indicated by a vertical line (|) to the left of the change.

SC24-6269-74, z/VM 7.4 (September 2024)

This edition supports the general availability of z/VM 7.4. Note that the publication number suffix (-74) indicates the z/VM release to which this edition applies.

SC24-6269-73, z/VM 7.3 (December 2023)

This edition includes terminology, maintenance, and editorial changes.

SC24-6269-73, z/VM 7.3 (September 2022)

This edition supports the general availability of z/VM 7.3. Note that the publication number suffix (-73) indicates the z/VM release to which this edition applies.

SC24-6269-02, z/VM 7.2 (March 2021)

[VM66201, VM66425] z/Architecture Extended Configuration (z/XC) support

With the PTFs for APARs VM66201 (CP) and VM66425 (CMS), z/Architecture® Extended Configuration (z/XC) support is provided. CMS applications that run in z/Architecture can use multiple address spaces. A z/XC guest can use VM data spaces with z/Architecture in the same way that an ESA/XC guest can use VM data spaces with Enterprise Systems Architecture. z/Architecture CMS (z/CMS) can use VM data spaces to access Shared File System (SFS) Directory Control (DIRCONTROL) directories. Programs can use z/Architecture instructions and registers (within the limits of z/CMS support) and can use VM data spaces in the same CMS session. For more information, see [z/VM: z/Architecture Extended Configuration \(z/XC\) Principles of Operation](#).

Information in the following topic is updated:

- [“Defining Your Diagnose Code” on page 48](#)

SC24-6269-01, z/VM 7.2 (September 2020)

This edition supports the general availability of z/VM 7.2.

Removal of PAGING63 IPL Parameters

The PAGING63 IPL parameter is removed since this IPL parameter blocks use of newer paging technologies (for example, Encrypted, EAV, HyperPAV, and HPF Paging) and has not been recommended for use since z/VM 6.4.

The following section has been updated: [Appendix K, “I/O Monitor User Exit,” on page 293](#).

CP Accounting Exit Information Moved

Information about CP Accounting Exit has been moved from [z/VM: CP Planning and Administration](#) to [Appendix E, “CP Accounting Exit \(Module HCPACU\),” on page 165](#).

Chapter 1. Introduction

This section introduces you to the CP Exit support (its terminology, concepts, and functions) and explains how the CP Exit support can be helpful to you.

z/VM allows you to customize its control program (CP) using IBM-supplied commands and configuration file statements. These commands and statements control the settings of various system functions. If you need to customize system functions that you cannot control using commands or statements, you can add non-IBM-supplied code (called "dynamically loaded CP routines") to CP. These additions allow you to extend the IBM-supplied system functions without requiring you to generate a new CP nucleus.

This document discusses the CP Exit support (also referred to as "CP Exits"), which allows you to make nondisruptive additions and deletions of dynamically loaded CP routines (non-IBM-supplied code). Using CP Exits, you can add and delete:

- CP commands
- Diagnose codes
- Locally-developed CP message repositories
- CP exit routines

Benefits of Using CP Exits

Using CP Exits gives you the ability to easily extend what your z/VM system already provides:

- Increased system availability by:
 - Eliminating the need to shut down and re-IPL z/VM to change user-executable code
- Reduced system generation errors by:
 - Eliminating the rework to local CP modifications, which are required because IBM changed the CP source files
 - Reducing the time, complexity, and frequency of SYSGENs by avoiding local CP modifications to the CP code linked during system generation
 - Eliminating SYSGENs for changes to local CP modifications
- Easy migration of existing z/VM customer base to the new releases
- Improved system programmer productivity by eliminating unnecessary tasks

Understanding the System Execution Space and CP Customization Methods

Before we can discuss how to dynamically add code to CP, you need to understand the layout of the address space in which CP executes. This address space is called the system execution space (SXS). Also, we need to compare the SYSGEN method of customizing CP (the old way) with the dynamic method of customizing CP (CP Exits).

While explaining the layout of the SXS and the two methods of customizing CP, we will use a simple example for the CP nucleus area of the SXS and the dynamic area of the SXS. In this example, we will ignore the following:

- Other parts of the SXS located above the dynamic area
- HSA (hardware system area)
- Other miscellaneous areas that are not relevant to our discussion

Figure 1 on page 2 and Figure 2 on page 3 show the CP nucleus at the bottom of the picture and the SXS dynamic area at the top. The SXS dynamic area is used for all of CP's dynamic storage requirements: CP control blocks, paging, and so forth.

Storage Addresses

A central storage address reference by CP is typically termed a Host Logical Address, or HLA. HLAs map the SXS. CP control blocks, the CP nucleus, and files loaded with CPXLOAD are all mapped in the SXS and are addressed by HLAs.

The SXS is not called CP Virtual Storage, even though the HLA undergoes hardware dynamic address translation (DAT) to generate a Host Real Address (HRA) and address prefixing in order to generate a Host Absolute Address (HAA). You can ignore HRAs and HAAs unless the need for such an address is explicitly stated. For example, the HRA and HAA would be used for hardware addresses like CCW addresses. You can ignore CP Virtual Storage unless the need for such an address is explicitly stated. Generally, write your programs and refer to CP control blocks just like always.

All references to storage addresses in this document are to HLAs unless otherwise stated.

Customizing CP Using the SYSGEN Method

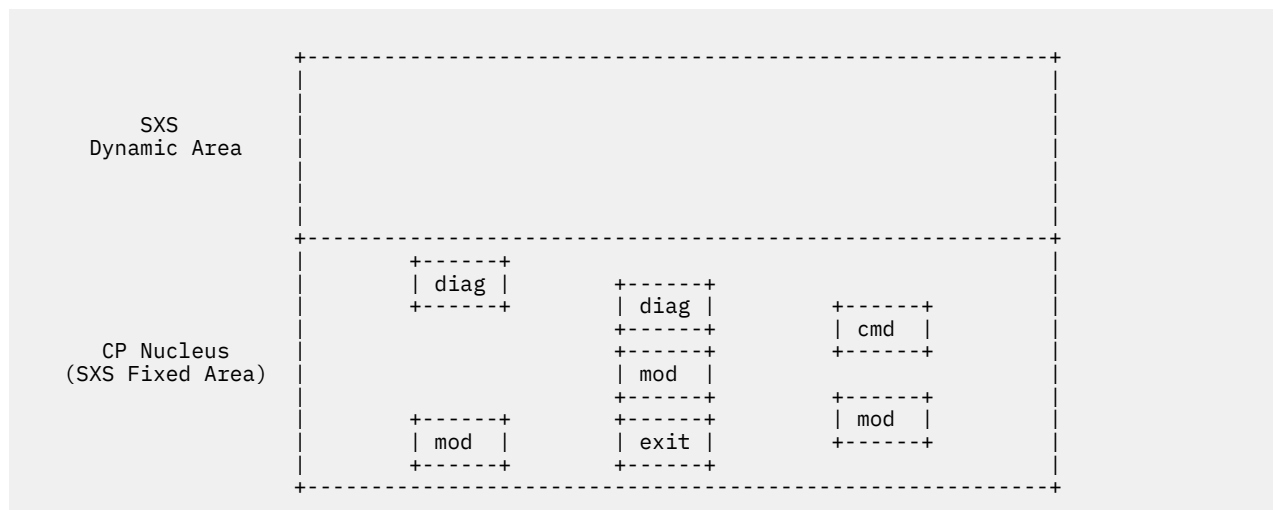


Figure 1. Layout of the SXS Using the SYSGEN Method

When you add your code to CP using the SYSGEN method, it gets placed in the CP nucleus, among all of the IBM-supplied CP code. To add your code in this manner, you must make local modifications to the CP source files. The SYSGEN handles the linkage considerations, such as resolving addresses.

This method is unpleasant because you do not own or control the CP source files. Every time IBM makes a change to a source file, that change has the possibility of causing you to rework your code. For example, you must contend with:

- Changes to module names
- Changes in sequence numbering
- Changes to internal design
- Modules whose source is not distributed outside IBM
- Sparse documentation

At times, this unpleasantness can confront you when you apply service within a release. After making your coding changes, you must regenerate the CP nucleus, shut z/VM down, and re-IPL, which causes an **unwelcome** loss of z/VM service.

Customizing CP Using the CP Exit Method

Using the CP Exits support discussed in this document, you can dynamically add your CP command routines, Diagnose code routines, CP exit routines and local message repositories to CP.

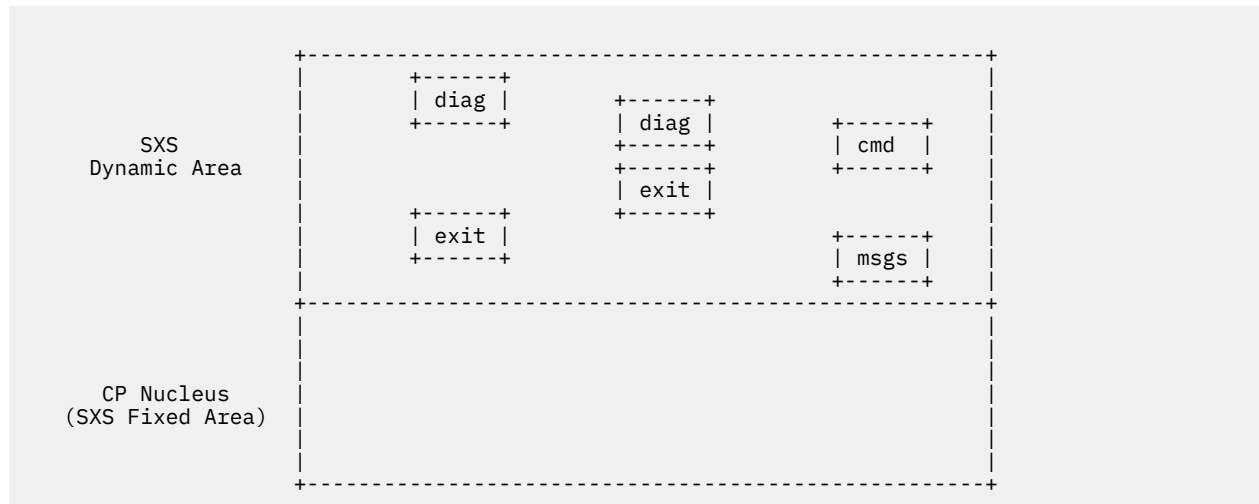


Figure 2. Layout of the SXS Using the CP Exits Method

Unlike the SYSGEN method, your code and data files are stored in the SXS dynamic area, not the CP nucleus. This greatly reduces the possibility that you will have to rework your code because IBM makes internal changes in CP source files. Your code and data files are easily added, removed, or replaced at your convenience, without disruption to z/VM service.

Note: The rest of this document concentrates on how to customize CP using the CP Exit method. The SYSGEN method will be mentioned only as a contrast.

Loading Files into the SXS Dynamic Area

To place your code and data files into the SXS dynamic area, use CPXLOAD, which is available as a configuration file statement and as a command. CPXLOAD enables you to load almost any text file that you could have loaded into the CP nucleus with a SYSGEN.

In order to dynamically load your files, CPXLOAD:

1. Reads your executable code and data files from the parm disk (or if available, from any CP-accessed disk).
2. Places the data into the SXS dynamic area.
3. Interconnects it with tables in the CP nucleus so that your executable code looks like it was there all along (that is, it looks like it was bundled together with the rest of the CP nucleus during a SYSGEN).
4. (Optionally) Invokes a control entry point to initialize the newly-loaded code. This may involve tasks such as setting up tables, acquiring storage, or defining locks. Refer to [Chapter 3, “Creating a Dynamically Loaded Routine,”](#) on page 11 and [Chapter 4, “Loading Dynamically into the System Execution Space,”](#) on page 33 for more information regarding the CONTROL and NOCONTROL operands of CPXLOAD.

After CPXLOAD completes, your executable code becomes part of CP without interrupting CP (or the rest of your z/VM system).

Unloading Files from the SXS Dynamic Area

In addition to being able to load your routines, you can also unload the routines you dynamically loaded (with CPXLOAD) using the CPXUNLOAD command. (CPXUNLOAD does not have a configuration file statement counterpart.) CPXUNLOAD removes the necessary linkages so that the system can continue

to operate without interruption. As with CPXLOAD, you can invoke an optional control entry point to allow your routines to clean up before the actual files are unloaded.

Creating, Controlling, and Redefining CP Commands and Diagnose Codes

You can assign the code that you dynamically load to a new CP command or to a Diagnose code. After loading the code, you must define the new command or Diagnose code to CP so that CP has the necessary control information to execute your command or Diagnose code. To define a new command, use the `DEFINE COMMAND` or `DEFINE CMD` command. To define a new Diagnose code, use the `DEFINE DIAGNOSE` command. Without these commands, you would have to use the `SYSGEN` method of system modification: coding the `COMMD` and `DIAGNOSE` macros, reassembling the data modules, regenerating the CP nucleus, and reIPLing the new system.

You can also redefine ("modify") existing CP commands and Diagnose codes. With this feature, you can dynamically change the user privilege class assigned to a CP command or Diagnose code. You can also change the entry point associated with the CP command or Diagnose code and thereby change the function.

Creating, Controlling, and Calling CP Exits

In addition to customizing CP commands and Diagnose codes, you can modify a code path by inserting code into an IBM-supplied module.

Using CP exit points allows you (or IBM) to define code transfer points. When enabled, these CP exit points transfer control to a CP exit routine. This minimizes the footprint of the user code in the IBM-supplied module, thus reducing the chance for conflicts with IBM service or new release support. CP exit points are defined during the expansion of the `HCPXSERV` macro at module assembly time.

The `HCPXSERV` macro defines the CP exit point by assigning an exit number to the point in the code at which the `HCPXSERV` macro expands. You can associate your user routines (entry points) with the CP exit point using the `ASSOCIATE EXIT` command or statement. The CP exit point processing code detects when a CP exit point is enabled and transfers control to the CP exit routine. Upon return, the routines that you associated with the CP exit point can set a return code to affect subsequent processing within the IBM-supplied module.

You can also define CP exit points dynamically using the `DEFINE EXIT` and `MODIFY EXIT` commands, or the `DEFINE EXIT` and `MODIFY EXIT` configuration statements. A dynamic exit point behaves just like a formally-defined exit point, except that its ability to influence subsequent processing in the module containing the exit point is limited.

Creating, Controlling, and Using Local CP Message Repositories

The code that you add to CP may need to generate messages. Just as adding code to the IBM-supplied modules left you susceptible to possible conflicts with IBM service changes and release changes, so does adding messages to the IBM-supplied CP message repository. To avoid such conflicts, you can create a local CP message repository and identify it to the system using the `ASSOCIATE MESSAGES` or `MSGSGS` command or configuration file statement. This allows your code to display messages from your own message repositories and eliminates the possible conflicts between your messages and the IBM-supplied messages in the CP message repository.

You can also use this function to replace an existing IBM-supplied CP message with one that is unique to your installation. CP allows you to create and associate a local message repository with the system so that messages in your repository override the messages in IBM's repository.

Chapter 2. Migrating Your Local Modifications

This section describes some common techniques that you can use to change your local modifications to IBM-supplied code into CP exit points.

Local modifications to IBM-supplied code can vary greatly in scope and size. You may be managing several small modifications that are scattered throughout a series of CP modules. Or, you may have added an additional function to an existing CP routine, CP command handler, or IBM-supplied Diagnose code. And finally, you may have written your own routine that, along with a few required source modifications to existing CP routines, provides a completely new command for your users or a new Diagnose code function for your applications.

The work required to manage source modifications to IBM-supplied code can be extensive. Installations that currently support local source modifications will want to migrate those changes in order to take advantage of the dynamic capabilities that are available for customizing your system. In some cases, very little work may be involved. In others, you may need to rework your local modification significantly to better fit into the dynamic arena. In particular, you will want to consider:

- Isolating your code additions from existing CP code
- Eliminating existing source modifications whenever possible
- Modifying your own routines so that they can be dynamically loaded on your system

Techniques for Isolating Source Modifications

There are significant advantages to isolating your existing changes to CP source code into your own modules. Removing local changes to IBM-supplied source code drastically reduces the need to rework those changes when IBM source code changes. In addition, removing your code changes from existing CP modules takes you a step closer to making your functional change dynamic.

Exploit an IBM-Defined CP Exit Point

A CP exit point defines a location in IBM-supplied code where a CP routine will give control to user routines. The user routines can perform additional tasks before returning control back to the CP routine. The IBM-defined CP exit points are listed and described in [Appendix A, “IBM-Defined CP Exit Points,” on page 89](#). Each one has defined entry conditions, specific parameter list requirements, and a set of exit conditions. By exploiting an available IBM defined CP exit point, you may be able to completely eliminate your source modifications to an existing CP routine.

Once you determine that there is a CP exit point that meets your needs, examine the functional changes that your local modification now provides. Your goal should be to delete changes to an existing source module and isolate the code changes to a separate module of your own. The local modifications might have to be enhanced so that the defined input and output parameters for the exit point are met.

As an example, suppose that you have a local modification to the CP initialization function. Because of the ramifications associated with performing a CLEAN start, you have decided to provide more control over the type of start that an operator can initiate. You may have added some code to existing CP source modules that prevents an operator from requesting a CLEAN start unless an extra piece of information is supplied when prompted.

Such a source modification would be a candidate for migration to the IBM defined CP exit point 00E1. This exit is driven after the operator responds to the initialization prompt, but before CP interprets the operator's input.

Once you have isolated the code change to handle the additional validation, you can use configuration file statements (CPXLOAD, ASSOCIATE EXIT, ASSOCIATE MESSAGES, ENABLE EXIT) to make use of this exit point and to activate your functional change dynamically.

Create Your Own Exit Point

There may be times where you find that you have a functional change to an existing CP routine and, after reviewing the list of available CP Exit points, you find there are none that meets your needs. This does not prevent you from migrating your source modifications so that they lend themselves to dynamic customization. Although in this instance you may not be able to completely eliminate your source modification, your goal should be to minimize your change to the IBM-supplied code as much as possible and to isolate the major portion of your changes to your own module.

Once you've isolated your functional changes, you could use the HCPXSERV macro to define your own exit point within a CP source module. The HCPXSERV macro is discussed in [Appendix F, “CP Exit Macros,” on page 169](#).

Instead of adding a formal exit point with the HCPXSERV macro, you can define an exit point dynamically using the DEFINE EXIT command. This capability is especially useful if you are not sure where to place an exit point. You can create one dynamically and try it out to see if its placement is appropriate. If you find that the exit point is not in the right place, you can use the MODIFY EXIT command to change its location or to alter the list of parameters that your exit routine receives.

Once you determine where to place an exit point in this way, it might be prudent to use the HCPXSERV macro to create a formal exit point. That way, changes that are made to the CP module through the service process or by new releases of z/VM are less likely to affect the continued operation of your customizations.

However, you may decide to always install your exit points dynamically. For example, if you simply want to gain control whenever some entry point is called, you may be able to create dynamic exit definitions that rarely need to be changed. In this case, you can use DEFINE EXIT configuration file statements to do this every time the system is IPLed.

Dynamic exits provide a convenient way to collect diagnostic or other information or to handle many situations in which the flow of control of a CP module does not need to be changed extensively. As explained in [“Standard Return Codes” on page 94](#), a dynamic exit has only limited ways to affect the flow of the module that contains the exit point. Of course, in some situations a judiciously placed exit point can allow an exit routine to alter conditions so that the flow of control is affected, but this is not always possible and might reduce the long-term maintainability of your customization. Where dynamic exits do not meet your needs, use formal exits.

In general, you should not create exits in CP module code paths where any of the following conditions hold:

- PFXTMPSV or PFXBALSV is in use.
- A loss of control cannot be tolerated.
- R11 does not point to a VMDBK.
- Early in system initialization.
- Late in system termination.
- Performance-sensitive code is executing.

Front-end an Existing Diagnose Code

Certain IBM-supplied Diagnose codes provide multiple services selected by a subcode passed to the Diagnose code routine. At some point, you may decide to extend the use of an IBM-supplied Diagnose code by creating a local modification in order to add a new subcode. To do so would have required source changes to the IBM-supplied module for the Diagnose code. This source code change, of course, is what you want to avoid.

An alternative approach might be to front-end the IBM Diagnose code routine with your own. In this way, you can add functional changes but avoid source modifications to IBM code.

Suppose that you want to add subcode X'xx' to DIAGNOSE code X'14'. The logic for your new routine would be something like this.

```
enter
get the subcode
select
  when( subcode = X'xx' )
  then do
    process it
  end
  otherwise
  do
    call the original diag x'14' routine, HCPDRDER
  end
end
return
```

Once you have isolated your functional change, you can use configuration file statements or CP commands (CPXLOAD, MODIFY DIAGNOSE) to provide your functional change dynamically. In particular, MODIFY DIAGNOSE can be used to change the entry point name that will be given control to handle the Diagnose code. This enables you to direct control to your new routine which would then call the IBM-supplied routine if appropriate.

Front-end an Existing CP Command

You may have local modifications on your system that add function to an existing CP command. Again, you need to examine the code changes that you have made with the thought of isolating your change from CP source code.

Suppose that you wish to perform additional checking for some command operand, for example, the SYSTEM operand on a PURGE RDR command. Once again, front-ending an IBM routine enables you to perform additional processing while avoiding changes to CP source code. In your routine, you would parse the command and perform whatever additional processing you wish. If all was acceptable, you would call the original IBM routine. The logic for your new routine would be something like this.

```
enter
save GSDBK fields
parse the command, performing necessary validation
if passed,
  then call the original command routine
  else reject the command
exit
```

Once you have isolated your functional change, you can use configuration file statements or CP commands (CPXLOAD, MODIFY COMMAND) to provide your functional change dynamically. In particular, MODIFY COMMAND can be used to change the entry point name that will be given control to handle the command. This enables you to direct control to your new routine which would then call the original IBM routine if appropriate.

Eliminating Source Modifications

There are ways to eliminate many of the small source modifications that you may have made as part of providing new commands, new Diagnose codes and new routines for CP.

Command Table Updates

Prior to VM/ESA® version 2, adding a new command to the system generally involved using the COMMD macro to create a command table entry to one of the following CP source modules:

- HCPSET for a new SET subcommand
- HCPQUY for a new QUERY subcommand
- HPCOM for other commands

The source modifications you have made to the command tables can be eliminated by using the `DEFINE COMMAND` command or configuration file statement. This will dynamically make changes to the system so that you do not have to code static changes in the CP routines.

Diagnose Code Table Updates

Prior to VM/ESA version 2, adding a new user Diagnose code to the system generally involved making an update to one of the following CP source modules:

- HCPHVB for handling a Diagnose code in the user range
- HCPHVC for handling a Diagnose code in the IBM range
- HCPDGN for defining a Diagnose code in the IBM range

The source modifications you have made to the Diagnose code routines can be eliminated by using the `DEFINE DIAGNOSE` command or configuration file statement. This will dynamically make changes to the system so that you do not have to code static changes in the CP routines.

Defining Linkage Attributes for New Modules

You generally define the linkage attributes for your new module by making a source modification to the IBM-supplied `HCPMDLAT` MACRO. To eliminate the need for this source modification, you can define a component ID to which your changed routines can belong and make use of the alternate MDLAT support. For additional information on using the `HPCMPID` macro to define a component ID, see [Chapter 3, “Creating a Dynamically Loaded Routine,” on page 11](#). For additional information on coding and using an alternate MDLAT see, [Chapter 3, “Creating a Dynamically Loaded Routine,” on page 11](#).

Selecting a Name for New Modules

For new modules that you may have previously created for your own use, you most likely selected a module name that begins with the characters HCP. HCP is the component ID that is associated with CP modules that IBM ships. Although you can certainly choose to continue to use this naming convention, you will still have to contend with any naming conflicts that arise should the module name you have selected be used by IBM in the future. To avoid the need to rename routines in the future, you may want to consider defining your own 3 character component ID by using the `HPCMPID` macro. You can then use your own component ID as part of the name you select for your routine. For additional guidelines on naming new routines, see [Chapter 3, “Creating a Dynamically Loaded Routine,” on page 11](#).

Changes to the System Common Area

The System Common Area (SYSCM) contains system-wide values such as counters and pointers that various CP functions use. You may have made source modifications to SYSCM in order to share information between your own routines. You may be able to remove your dependency on these source updates by using the Component ID Block (CMPBK).

This control block is designed for use by user code. The `HCPXSERV` macro provides functions you can use to allocate, deallocate and serialize the use of this control block. For more information on the functions provided by the `HCPXSERV` macro for use with CMPBKs, see [Appendix F, “CP Exit Macros,” on page 169](#).

Changes to the CP Message Repository

You may have source modifications to the CP message repository. This can vary from a simple change, such as rewording existing message text, to something more complicated that would include changes to the number and order of substitution variables. You may also have placed new messages in the CP message repository that other local modifications utilize.

By creating your own local message repositories, you can eliminate your source modifications to the IBM-supplied message repository. Additional information on creating message repositories can be found in [Chapter 7, “Defining and Using CP Message Repositories,” on page 51](#) as well as in [Appendix H, “Understanding the CP Message Repository,” on page 209](#).

Dynamically Loaded Routines

Routines that are dynamically loaded are treated like IBM-written routines. This means modules you have already developed that use standard HCPCALL linkage conventions and have dynamic saveareas can often be dynamically loaded without requiring significant changes. On the other hand, routines you have developed that are entered by some means other than a dynamic call or that use static saveareas will probably require significant rework.

As a starting point, review [Chapter 3, “Creating a Dynamically Loaded Routine,” on page 11](#), which describes, in general, what you need to do to create dynamically loaded CP routines. If you already have your own CP module, and you are planning to convert this module so that it can be dynamically loaded, here is a list of common areas to focus on:

- HCPENTER should specify CALL and SAVE=DYNAMIC.
- HCPCALL should specify TYPE=INDIRECT and CALLFAIL on all calls to routines that you plan to dynamically load.
- Remove adcon references in the CP nucleus to fields in routines you plan to load dynamically.
- Use HCPCONSL to display messages from a message repository.
- Use HCPTRANS to translate virtual addresses to their logical, real, or absolute counterparts. Do not use HCPTRANS to get the Host Real Address from a Host Logical Address; use the LRAG instruction.

Chapter 3. Creating a Dynamically Loaded Routine

This section describes some common issues you should consider when creating a dynamically loaded routine. As you design your dynamically loaded routine, there are a number of issues that will arise that you must decide how to handle. The following is a list of issues for you to consider. After this list, the individual issues are discussed in more detail to help you make your decision.

The primary question you need to ask yourself is: "How will my routine interface with the rest of CP?" That is:

1. From where will I get my input data?
 - Input in registers
 - Input located by addresses in registers
 - Input in predefined locations in host storage
2. To where will I put my output data?
 - Output in registers
 - Output located by addresses in registers
 - Output placed into predefined locations in host storage
3. What type of addressing mode should I use?
 - 31-bit or 64-bit
4. What, if any, CP Services should I use?
 - HCPCMPID
 - HCPPROLG
 - HCPENTER
 - HCPEXIT
 - HCPCALL
 - HCPLCALL
 - HCPLENTR
 - HCPLEXIT
 - HCPCONSL
5. Will I be calling other dynamically loaded routines from this routine?
 - HCPCALL TYPE=INDIRECT
6. How big can I make my routine?
 - 4 KB or more?
7. What should I name my routine?
 - HCPCMPID
 - HCPPROLG
 - HCPENTER
8. What linkage attributes should I specify for my routine?
 - HCPCMPID
 - xxxMDLAT
9. How are addresses resolved?
 - Address constants in the nucleus

10. Can I add my own CP IUCV System Service?
11. What does the CPXUNLOAD ASYNCHRONOUS operand mean?
12. What does the CPXLOAD CONTROL operand imply?
13. What happens if I make a programming error?

Input Data

Deciding how to provide input data to any routine is an exercise where you attempt to decide between simplicity and flexibility. Typically, passing data in registers is the simplest of mechanisms. There is no need for both the calling routine and the called routine to use an elaborate parameter list.

However, passing data in registers, while simple, is also quite restrictive. The number of registers available is quite limited, which in turn limits the amount of data and constrains data formats. For example, you might decide to pass in a register to another routine the 4-byte disk block number of an FBA (Fixed Block Architecture) disk. Later, if you extend the routine's capabilities to include support of CKD (Count-Key-Data) disks, you would need to pass a 5-byte data item (2-byte cylinder, 2-byte track, 1-byte record). The interface between the routines must now be made radically different, requiring one register for an FBA disk, and 2 registers for a CKD disk. Such interface differences readily lead to programming errors. Moreover, if this interface (1 register versus 2 registers) were being changed for a large number of routines, you might find yourself spending much time modifying interfaces between many related routines for the purposes of safety and consistency, even though these other routines might not have been directly affected by the change to the new interface.

Passing addresses of data in registers is perhaps more flexible. Then, if the length or the type of data changed, as in the example above, the interface would not. The address would simply point to a longer data item.

You can further extend the parameter passing flexibility with a slightly more complicated interface where one register is used to point to a list of addresses, each of which then points to a data item. Such an interface is slightly more complicated, but readily extended as needs change. To help manage this complexity, you can provide mapping macros and DSECTs that all related routines would include. Management of changes to the interfaces would then be simplified by changes to the common mapping macros and DSECTs.

Occasionally, you find that the data items that the routine will use are in predictable locations in storage. For example, the data may be located in a system control block (say, in SYSCM, the System Common Area or in a VMDBK). In such a case, the routine need not be passed anything, because it can locate the data itself. An example of such a routine is HCPSCCFD, which parses the next token in a CP command. When called, HCPSCCFD knows that the area to work with is located by the VMDBK. The calling routine does not need to pass to HCPSCCFD any additional data. (We ignore the fact that the address of the VMDBK is in register R11, because this fact is common to most of CP.) This interface is simple but inflexible. If you need to extract a token from some other area that does not contain the active CP command, you cannot use HCPSCCFD. Another routine must be used, in this case, HCPSCCFG. Is it better to provide 2 routines (HCPSCCFD and HCPSCCFG) that do exactly the same thing but get the address of the data area from different places? You be the judge.

When deciding how to pass data to other routines, also consider the ultimate source of the data. If the data can be relatively stable, consider using an external repository (for example, a CMS file) to store the data. By storing the data in a way that affords you the ability to use CMS tools (XEDIT, COPYFILE, SENDFILE, RECEIVE, etc.), you may be able to assist the distribution of the data to other systems, which may be beneficial. For more information on standard input data provided to CP exit routines, see [Appendix A, "IBM-Defined CP Exit Points," on page 89.](#)

Output Data

All of the discussion on Input Data applies here, as well. But in addition, you may need to consider whether the calling routine should allocate storage for the output data, or whether the called routine should. And, if the called routine should, how it should return the address of the newly-allocated storage to the calling routine.

Addressing Mode

The hardware on which z/VM runs provides an addressing mode called Access Register mode (AR mode). CP provides macros that assist you when writing routines that use AR mode. In AR mode, Dynamic (or, automatic) Address Translation (DAT) of AR-qualified addresses is a standard hardware facility. AR mode is a requirement for accessing central storage when only the Host Real Address is available.

Address constants are created directly when you write the source code for a routine and use the appropriate DC (Define Constant) statement or indirectly when certain macros (for example, HCPCALL) are expanded during execution of the IBM High Level Assembler. An address constant is resolved during SYSGEN by HCPLDR when it builds the CP nucleus, during which time it determines the locations of symbols to which the address constants refer, and during dynamic loading by CPXLOAD when it loads a file, during which time it does essentially the same thing as HCPLDR. After HCPLDR finishes the SYSGEN process, any address constant whose target cannot be found is ignored and an error message is displayed; nothing is remembered regarding that unresolved address constant. After CPXLOAD finishes its load process, any address constant whose target cannot be found is remembered so that it can be resolved by a later CPXLOAD operation.

Addresses in CP for control blocks, modules, and other "CP things" are generally Host Logical Addresses. Addresses in CCWs, SCMBKs, Control Registers, and other "hardware things" are generally Host Real Addresses or Host Absolute Addresses. AR mode using the ALET provided in field PFXRSAL can be used to touch central storage using a real address.

Routines loaded by CPXLOAD are treated exactly as if they were part of the CP nucleus. Address constants to data items in a routine loaded by CPXLOAD are resolved to Host Logical Addresses by the dynamic loading process.

Attributes

Modules and their entry points have a number of attributes, all of which are used by linkage macros to generate proper instruction sequences. These attributes are specified in the HCPMDLAT macro, or the equivalent Component ID macro xxxMDLAT.

Register Styles

A CP module may have one of two register styles:

- ShortReg (the default)
- LongReg

ShortReg modules use 32-bit registers and LongReg modules use 64-bit registers. This module attribute is declared by the MDLATENT macro. A module's entry points may use different styles.

Calls between modules of different styles are supported. The name and format of the save area block depends on the styles of the called and calling modules. ShortReg entry points use SAVBK COPY to map their register save areas. LongReg entry points use SVGBK COPY to map their register save areas. The SVGBK provides areas to save contiguous 64-bit general registers, and the SAVBK provides areas to save discontinuous parts of 64-bit registers. As a result, LM and STM instructions must be used with care.

Except under extraordinary circumstances, a ShortReg module must not reference or change the high-order four bytes of a large (64-bit) register. While in general a module's entry points use the same register style as the module itself, the two may be different (for example, to facilitate calls from modules that use another style). In this case, once the entry to the module has been effected and any parameters have been transformed, the bulk of the module's logic is expected to use the register style declared for the module. While this is not enforced, it is good practice.

Entry points that are invoked with an indirect call (HCPCALL TYPE=INDIRECT) must be ShortReg. This includes CP Exits and dynamically loaded modules that replace or define new commands and Diagnose functions.

Addressing Mode

Addressing Mode, or Amode, describes the settings of PSW bits 31-32. Module Amode attributes are:

Amode31

The module and its entry points run in 31-bit addressing mode.

Amode64

The module and its entry points run in 64-bit addressing mode. This requires the LongReg attribute.

AmodeVAR

The module is unpredictable as to whether it is in 31-bit or 64-bit addressing mode. A default entry point attribute is required. This requires the LongReg attribute.

AmodeINT

The module and its entry points run in 31-bit addressing mode, except that the addressing mode may be changed. However, the module will ensure that any addressing mode manipulation will not be exposed to any macro or to any other module.

There are also specific Entry Point attributes for addressing mode. See [“Entry Point Attributes” on page 14](#).

The CP default is **Amode31**, which means that 31-bit addressing mode is in effect everywhere.

To assist the programmer, macros will use the Amode attribute to generate code that does (or does not, depending on the situation) save and restore the addressing mode bits.

Translation Mode

Translation Mode, or Tmode, describes the settings of PSW bits 16-17. Module Tmode attributes are:

TmodeSTD

The module and its entry points run in Primary Space mode and do not alter ARs.

TmodeAR

The module and its entry points run in Access Register mode and do alter ARs.

TmodeVAR

The module is unpredictable as to whether it is in AR mode or Primary Space mode. A default entry point attribute is required.

TmodeINT

The module and its entry points run in Primary Space mode, except that the ARs may be used. However, the module will ensure that any manipulation of ARs or of the PSW bits 16-17 will not be exposed to any macro or to any other module.

There are also specific Entry Point attributes for translation mode. See [“Entry Point Attributes” on page 14](#).

Historically, CP ran very nearly in the **TmodeINT** style, which meant that only the module itself knew how the Access Registers were being used. Macros knew nothing, so no help was forthcoming in code generated by linkage macros.

Now, the CP default is **TmodeSTD**, which means that ARs are not altered by the module. This means that a module that is using ARs can rely on them not being altered by any subroutine that it calls. In fact, just like modifications to general registers, no modifications to Access Registers should be seen by any other module unless disclosed as a documented interface.

To assist the programmer, macros will use the Tmode attribute to generate code that does (or does not, depending on the situation) save and restore ARs so that the rule "no modifications to ARs will be exposed" is enforced across all linkage macros.

Entry Point Attributes

While a module's attributes are considered to be module-wide, each entry point may have different attributes. For example, a module may usually run with the TmodeAR attribute (that is, always in AR mode), but the module may have an entry point that is declared to be entered in Primary Space mode.

This entry point must be declared in xxxMDLAT as TmodeSTDent. The "ent" suffix distinguishes this as an "entry point attribute". Although most module attributes imply default entry point attributes, module attributes are not entry point attributes.

The entry point attributes for addressing mode, translation mode, and register style are:

Amode31ent

The entry is in 31-bit addressing mode.

Amode64ent

The entry is in 64-bit addressing mode.

AmodeVARent

The entry is unpredictable as to whether it is in 31-bit or 64-bit addressing mode. The programmer must write code to handle either case.

TmodeSTDent

The entry is in Primary Space mode and will not alter ARs.

TmodeARent

The entry is in Access Register mode and will alter ARs.

TmodeVARent

The entry is unpredictable as to whether it is in AR mode or Primary Space mode. The programmer must write code to handle either case.

ShortRegEnt

The entry will use only the low halves of the general registers. The entry must be Amode31ent.

LongRegEnt

The entry will use the entire 64 bits of the general registers.

The following table shows the implied default entry point attributes that correspond to module attribute settings and defaults:

Module Attribute	Implied Default Entry Point Attribute
Amode31	Amode31ent
Amode64	Amode64ent
AmodeVAR	(none)
AmodeINT	Amode31ent
TmodeSTD	TmodeSTDent
TmodeAR	TmodeARent
TmodeVAR	(none)
TmodeINT	TmodeSTDent
ShortReg	ShortRegEnt
LongReg	LongRegEnt

The module attributes are true everywhere throughout the module at every macro invocation, except for HCPENTER and HCPEXIT, where only the entry point attributes are assumed.

This is a very subtle but important point. A combination like TmodeAR TmodeSTDent means that the entry point will be in Primary Space mode, but outside of HCPENTER and HCPEXIT every macro will be expected to be in AR mode. The entry point's HCPENTER will generate code for Primary Space mode, but no SACF SACAR instruction will be generated by the HCPENTER macro. The programmer must ensure that this kind of mismatch is resolved at runtime by coding the necessary SACF instructions, both after HCPENTER and before HCPEXIT.

Environment at Enter and Exit

The runtime environment at entry must match that at exit. So a TmodeARent entry point will be in AR mode when the HCPENTER macro finishes and must be in AR mode when the HCPEXIT macro is executed. If you change the mode for any reason, you must reestablish the environment before HCPEXIT.

Using CP Services

Any CP service is available to a module that was loaded by CPXLOAD, as long as a simple HCPCALL--HCPEXIT protocol is used. If other mechanisms of transferring control are attempted (for example, HCPGOTO), results are officially "unpredictable". What this means is that such an environment is too complex to predict the outcome. Certain uses of HCPGOTO to transfer execution to the dispatcher (HCPDSP) and stacked CPEBKs to resume execution may, in fact, work. Only by exhaustive testing will you be able to determine this.

The discussion here of CP macros is not exhaustive. Only those operands that are considered the most likely to be used will be discussed.

Defining System Linkage to Your Routine

In order for the standard CP linkage macros HCPPROLG, HCPENTER, HCPCALL, and HCPEXIT to generate the correct assembler instructions in your routine, you must define the routine's linkage attributes. This is done by:

1. Specifying a HPCMPID macro at the beginning of your routine. The purpose of the HPCMPID macro is to specify the component ID of which your routine is a part. The component ID is then used to identify which xxxMDLAT macro (also called "alternate" MDLAT macro) is to be searched for your routine's linkage attributes.
2. Coding a xxxMDLAT macro. The format of the xxxMDLAT macro is much simplified over the format of IBM's HCPMDLAT MACRO. Here is an example of an alternate MDLAT macro called EXTMDLAT that defines your routine that is called EXT00E1:

```
&LABEL      MACRO
EXTMDLAT &EPNAME
MDLATHDR &EPNAME
MDLATENT EXT00E1,MODATTR=(MP,DYN),CPXLOAD=YES
MDLATTLR
MEXIT
MEND
```

The associated HPCMPID macro in your routine would be coded:

```
HPCMPID COMPID=(EXT)
```

3. Placing your xxxMDLAT macro where the assembler can find it when you assemble your routine. The easiest way to do this is to place the xxxMDLAT macro right in your routine as an "in-line macro definition". Because your routine should define the macro before it is used, it is best to put your xxxMDLAT macro as the first thing in your routine's source file.

Another place to put your xxxMDLAT macro is in one of the maclibs that the assembler uses when it assembles your routine. If you are using VMFHASM or VMFHLASM, place your macro in one of the maclibs that are specified on the MACS statement of the control file.

Discussion of the HCPPROLG Macro

The HCPPROLG macro establishes certain attributes for the entire module. Some of these attributes are:

- The module name

The label on the HCPPROLG establishes the module name, which is also the Control Section (CSECT) name.

- REUSABLE or REENTRANT or REFRESHABLE

The goal here should be REFRESHABLE. This means that CP could reload a copy of the module at any time without damage. This means that, once loaded, the module may never be altered. Unfortunately, any indirect linkage by HCPCALL causes the Indirect Call Request Block in the module to be changed.

Because REFRESHABLE may be impossible, the next best choice is REENTRANT. This implies that any number of concurrent tasks may be executing instructions in the module simultaneously. In a multiprogramming, multiprocessing system, this should be the case. REENTRANT tends to be interpreted as "unchanging". This is a good interpretation. However, techniques exist that allow changes to the module while retaining reentrancy. For simplicity, forget about this and just never change the module.

REUSABLE is the next least desirable. It means that only one task may be executing instructions in the module at a time. After that task exits the module, another may enter it. Such serialization could seriously hamper performance and throughput.

NONREUSABLE is the least desirable. It means that only one task may execute instructions in the module. After that task exits the module, no other may enter it.

Realistically, CP treats everything as REENTRANT, so you should code to that level.

- DIRECT or INDIRECT

HCPCALL=DIRECT is typical throughout all of CP.

HCPCALL=INDIRECT should be specified for any dynamically loaded module.

For information about the meaning of direct and indirect linkage, see ["Discussion of the HCPCALL Macro" on page 18](#).

- Base register

A module should need only one base register, and it must be R12. So, specify BASE=(R12) on your HCPPROLG macro.

- Copyright

Copyright information may also be specified on the HCPPROLG macro. If specified, instructions will be generated to store the copyright data in the resulting TEXT file so that anyone who looks at storage or at a system dump will see the copyright data.

To specify copyright year, use the COPYR= keyword. To add textual data to the copyright year, also include the COPYRID= keyword.

Here are some examples of HCPPROLG macros:

VREPOS	HCPPROLG ATTR=(REENTRANT),HCPCALL=INDIRECT, BASE=(R12)	X
JAMSAM	HCPPROLG ATTR=(REENTRANT),HCPCALL=INDIRECT, COPYR=(1995),COPYRID='My Copyright', BASE=(R12)	X X

Discussion of the HCPENTER Macro

The HCPENTER macro defines an executable entry point into the module. The name of the entry point is the label on the HCPENTER macro. This name must start with the same 6 characters as the label on the HCPPROLG macro, plus 1 or 2 additional characters.

Operands on the HCPENTER macro indicate what instructions should be generated for entry into the module:

- CALL

HCPENTER can describe a number of types of entry into the module. But all dynamically loaded modules are entered as a result of an HCPCALL macro. Always specify CALL as the first operand.

- SAVE=DYNAMIC

CP has two types of save area control blocks:

Defining System Linkage

- Static
- Dynamic

All dynamically loaded modules must use dynamic save area control blocks. Always specify SAVE=DYNAMIC as the second operand.

Here are some examples of HCPENTER macros:

```
VREPOSMS HCPENTER CALL,SAVE=DYNAMIC
JAMSAMD L HCPENTER CALL,SAVE=DYNAMIC
```

Discussion of the HCPEXIT Macro

The HCPEXIT macro defines the exit from a module. The name of the entry point is included in an operand of the HCPEXIT macro. Also, you can direct whether the current condition code should be preserved and returned to the caller. The operands include:

- EP=

The name of the entry point that was called and for which the exit and return to the caller is being made is specified by the EP= keyword. Multiple entry points can exit through a single HCPEXIT macro. If so, all would be included in the EP= list of entry points.

- SETCC=<YES|NO|0|1|2|3>

The SETCC= keyword indicates how the condition code should be treated.

- SETCC=YES, also the default, indicates that the present condition code should be preserved and returned to the caller.
- SETCC=NO indicates that the present condition code need not be preserved and returned to the caller. The condition code returned to the caller is unpredictable.
- SETCC=0 indicates that condition code 0 should be set and returned to the caller.
- SETCC=1 indicates that condition code 1 should be set and returned to the caller.
- SETCC=2 indicates that condition code 2 should be set and returned to the caller.
- SETCC=3 indicates that condition code 3 should be set and returned to the caller.

Using the condition code to return information to the calling program provides limited capability because the possible values are restricted to 4 values. This may constrain future extensions to a routine. Serious thought should be given to using return codes to return information to the calling program. To set a return code of zero for the caller, place the value zero into the field SAVER15:

```
SLR    R15,R15
ST     R15,SAVER15
```

Here are some examples of HCPEXIT macros:

```
HCPEXIT EP=VREPOSMS
HCPEXIT EP=(JAMSAMD L,JAMSAMD L),SETCC=NO
```

Discussion of the HCPCALL Macro

The HCPCALL macro can be used to call a routine by using its address or by using its name. These two methods are called "direct" and "indirect" linkage, respectively. Direct linkage is further broken down into "call-by-attribute" and "call-by-register".

HCPCALL TYPE=DIRECT

This is the typical method used within CP to call another routine. In fact, this is so typical that you never even specify TYPE=DIRECT. Part of the processing of the HCPCALL macro is to identify the attributes

of both the calling routine and the called routine, and then to generate instructions appropriate to the interface between the two routines. For example, the caller might have the attribute MP, meaning that it may execute on any processor in a multiprocessor system, while the called routine might have the attribute NONMP, meaning that it may execute only on the master processor. Because a switch from alternate processor to master processor may be needed, based on these attributes, the HCPCALL macro will generate instructions to go to the appropriate entry in HCPSVC to affect the linkage. A different set of attributes for the calling routine and the called routine would cause HCPCALL macro to generate instructions to go to a different entry in HCPSVC.

The following table shows the various linkage decisions that the HCPCALL macro must make. Which row is chosen could depend on such details as:

- The called routine's name
- The called routine's address
- The calling routine's name
- Keywords specified on the HCPCALL macro invocation

<i>Table 2. Various Linkage Decisions</i>		
From=	To=	Trace=
MP	MP	NO
MP	MP	YES
MP	NONMP	NO
MP	NONMP	YES
NONMP	MP	NO
NONMP	MP	YES
NONMP	NONMP	NO
NONMP	NONMP	YES

Legend:

From	- capability of caller
To	- capability of callee
Save	- type of SAVBK requested by callee
Trace	- trace entry is desired
MP	- multiprocessor capable
NONMP	- not multiprocessor capable

The decision by HCPCALL will be rather automatic if the called routine is specified by name or if the attributes of the called routine are explicitly provided to HCPCALL. If the routine name is provided, HCPCALL will use HCPMDLAT to identify the routine's entry point attributes, and generate code accordingly. Or, if no routine name is provided but its attributes are, HCPCALL will use the supplied entry point attributes to generate code. These mechanisms, where the entry point attributes can be deduced or are specified, are examples of "call-by-attribute".

If neither a name (so attributes cannot be inferred) nor attributes are specified, and only the address of the routine is supplied, HCPCALL will use a technique known as "call-by-register". HCPCALL will determine the proper linkage to use to call the routine based on a vector of addresses provided by the called routine. The only requirement is that the called routine be defined as an HCPENTER SAVE=DYNAMIC entry point.

Call-by-register can be used to call any SAVE=DYNAMIC routine. Call-by-register cannot be used to call any other routine. For these others, some form of call-by-attribute must be used.

Here are a few examples where TYPE=DIRECT is assumed by default:

1. HCPCALL HCPCVTBD

In this example, HCPCALL knows that the linkage requirements for HCPCVTBD requires a direct branch to HCPCVTBD, without the dynamic allocation of a SAVBK. Because HCPCALL can determine all of the attributes of the linkage, the proper linkage will be generated.

2. ``` L R15,=A(HCPCVTBD) HCPCALL (R15),NAME=HCPCVTBD ```

In this example, the address of HCPCVTBD is already in register 15, so HCPCALL can use it. And because HCPCALL knows that the routine to call is HCPCVTBD, HCPCALL can generate the remainder of the linkage instructions correctly. Because HCPCALL can determine all of the attributes of the linkage, the proper linkage will be generated.

3. ``` L R15,=A(HCPCVTBD) HCPCALL (R15),ATTR=(NONE,LEAF,LRegE,AmVARE,TmVARE) ```

Again, the address of HCPCVTBD is already in register 15, ready to use. Because the routine's attributes are specified, HCPCALL will generate linkage instructions correctly.

Notice that the routine's attributes are entry point attributes, not module attributes. 'LRegE' is correct, but 'LReg' would be wrong. LongReg is not an entry point attribute.

4. ``` L R15,=A(HCPCVTBD) HCPCALL (R15) ```

Again, the address of HCPCVTBD is already in register 15, ready to use. Because no attributes can be inferred, and none are specified, HCPCALL will expect SAVE=DYNAMIC and will look for the vector of addresses in order to determine what linkage to use. Because HCPCVTBD is not SAVE=DYNAMIC, no such vector exists. The wrong linkage protocol will result, and CP will certainly crash.

Always ensure that HCPCALL can determine the proper entry point attributes of the routine that you are calling.

HCPCALL TYPE=INDIRECT

This is another typical method used within CP to call a routine. TYPE=INDIRECT is used by CP to call routines for processing the following:

- Commands, if the routine was dynamically loaded
- Diagnose codes, if the routine was dynamically loaded
- Exits, for all exit routines

Just like HCPCALL TYPE=DIRECT discussed above, HCPCALL TYPE=INDIRECT can generate the proper linkage if the routine name is known. If the routine is in the nucleus, then HCPCALL can convert the INDIRECT request to a DIRECT request and avoid the additional overhead of the INDIRECT request. If the name is not supplied, or the name is not in the nucleus, then HCPCALL will generate the indirect linkage.

The protocol for indirect linkage is that register 15 contains the address of the Indirect Call Request Block (ICRBK) that names the routine that you want to call. CP will use the contents of the ICRBK to locate the linkage attributes for the called routine, and proceed accordingly. By specifying the name of the routine to call, you can ensure that HCPCALL will generate the proper direct linkage to a routine in the nucleus by using its address, or will generate the proper indirect linkage using an ICRBK.

Because the purpose of the indirect linkage is to provide a safe calling sequence even if the routine being called does not exist, an additional keyword is always required whenever TYPE=INDIRECT is specified. That additional keyword is the CALLFAIL= keyword. This keyword indicates the label for HCPCALL to branch to if the calling sequence cannot be completed. CALLFAIL= is not for the situation where the calling sequence was successful and the called routine returned an error indication but is for the situation where the called routine was not entered. If this occurs, then register 15 can be examined to determine why the call attempt failed:

4

The target could not be called because:

- It was marked for CPXUNLOAD.
- It had not completed CPXLOAD.
- The address translation failed.

8

Control blocks used by HCPCALL did not match; for example, the ICRBK and its ICLBK had different entry point names.

12

The attempted call was required to be performed without any loss of control of the processor, but a loss of control of the processor would have been necessary.

Here are a few examples where TYPE=INDIRECT is specified:

1. `HCPCALL HCPCVTBD,TYPE=INDIRECT,CALLFAIL=errlabel`

In this example, HCPCALL knows that the linkage requirements for HCPCVTBD require a direct branch to HCPCVTBD, without the dynamic allocation of a SAVBK. Because HCPCALL can determine all of the attributes of the linkage, the proper linkage will be generated. Even though TYPE=INDIRECT was specified, HCPCALL knows that an indirect linkage is not necessary and will generate a direct linkage.

2. `L R15,=A(HCPCVTBD)
HCPCALL (R15),NAME=HCPCVTBD,TYPE=INDIRECT,CALLFAIL=errlabel`

In this example, the address of HCPCVTBD is already in register 15, so HCPCALL can use it. And because HCPCALL knows that the routine to call is HCPCVTBD, HCPCALL can generate the remainder of the linkage instructions. Because HCPCALL can determine all of the attributes of the linkage, the proper linkage will be generated. Even though TYPE=INDIRECT was specified, HCPCALL knows that an indirect linkage is not necessary and will generate a direct linkage.

3. `L R15,=A(HCPCVTBD)
HCPCALL (R15),TYPE=INDIRECT,CALLFAIL=errlabel`

In this example, HCPCALL does not know the name of the routine to call. Therefore, HCPCALL has no other choice but to generate in indirect linkage. Unfortunately because this example specifies an indirect call, HCPCALL requires that register 15 contain the address of the ICRBK. Register 15 does not; register 15 contains the address of the HCPCVTBD entry point. This mismatch will likely result in a CP abend.

In general, always tell HCPCALL the name of the routine that you are calling.

Usage Note for HCPCALL

HCPCALL ensures that no extra positional parameters are specified. If any are found, the following message is displayed:

```
MNOTE 8, 'Extra positional parameter: pppppppp'
```

If *pppppppp* is blank, this message could indicate that incorrect continuation was used on the HCPCALL invocation.

Discussion of the HCPLCALL Macro

The HCPLCALL macro provides a convenient mechanism to call a local routine inside a module. Entry to a local routine typically uses a dynamic save area block. Because HCPLCALL uses the services of HCPCALL, all of the discussion under HCPCALL applies here as well.

Here are some examples of HCPLCALL macros:

Defining System Linkage

1. `HCPLCALL SQUEEZE`

HCPLCALL knows the linkage requirements for all local routines. Register 15 will be loaded with the address of SQUEEZE and the linkage will be performed.

2. `LA R15,SQUEEZE
HCPLCALL (R15)`

In this example, the address of SQUEEZE is already in register 15, so HCPLCALL can use it.

3. `L R15,=A(SQUEEZE)
HCPLCALL (R15)`

In this example, the address of SQUEEZE is in register 15, and HCPLCALL will use it.

In general, always tell HCPLCALL the name of the routine that you are calling.

Discussion of the HCPLINTR Macro

The HCPLINTR macro defines an executable entry point in the module for a local routine. The name of the entry point is the label on the HCPLINTR macro. This name may start with any valid assembler language label. It does not need to be related to the HCPPROLG or HCPENTER macros.

If no operands are specified on the HCPLINTR macro, HCPLINTR will generate entry linkage for `HCPLINTR CALL,SAVE=DYNAMIC`. For a stacked goto to a local entry where there is no need to define an external symbol, HCPLINTR GOTO should be used.

Here are some examples of HCPLINTR macros:

```
LOOKUP    HCPLINTR
Q         HCPLINTR
```

Discussion of the HCPLEXIT Macro

The HCPLEXIT macro defines the exit from a local routine entered by HCPLINTR. The name of the entry point is not mentioned. The HCPLEXIT macro expands as if the EP= keyword was specified. SETCC= is the only operand HCPLEXIT accepts. For information about the SETCC= operand, refer to [“Discussion of the HCPEXIT Macro” on page 18](#).

Here are some examples of HCPLEXIT macros:

```
HCPLEXIT
HCPLEXIT SETCC=NO
```

Discussion of the HCPCONSL Macro

You should always use HCPCONSL to write a message or a response or to read input from the terminal. There are three operations that you can request from HCPCONSL.

In the discussions that follow, "<something>" means that you have specified a value, but not "NONE" or any equivalent value.

1. Write data or read data or both?

- Write: HCPCONSL WRITE
- Read: HCPCONSL READ
- Both: HCPCONSL PROMPT

This discussion will address only WRITE operations.

2. Writing what kind of message?

HCPCONSL can generate different types of messages. CP will direct the message based on its type. But sometimes, you need to direct the message because CP does not fully meet the challenge. This extra processing is true for the IMSG type of message.

- Using the user's EMSG setting

```
HCPCONSL WRITE,
      DATATYPE=EMSG
```

Using the value set for EMSG, CP will format and direct the message:

- For SET EMSG ON, CP will format the message to contain both the error message heading and the error message text.
- For SET EMSG OFF, CP will not generate the message.
- For SET EMSG TEXT, CP will format the message to contain only the error message text.
- For SET EMSG CODE, CP will format the message to contain only the error message heading.
- For SET EMSG IUCV, CP will format the message to contain both the error message heading and the error message text.

- Overriding, partially, the user's EMSG setting

```
HCPCONSL WRITE,
      DATATYPE=FULLEMSG
```

CP will format the message to contain both the error message heading and the error message text.

- Using the user's IMSG setting

```
HCPCONSL WRITE,
      DATATYPE=IMSG
```

Here is a situation where you need to do some extra work to make sure that the message is not generated when it should not be. Even if SET IMSG OFF is in effect, CP will still generate the message. It is up to you to check for IMSG OFF and, if so, not generate the message in the first place. To check the IMSG setting, test the VMDBK byte VMDMLVL for the flag VMDMIMSG. If on, then the IMSG should be generated.

```
TM      VMDMLVL,VMDMIMSG      Is IMSG wanted?
BZ      skip                  ..off, the answer is no
HCPCONSL WRITE,
      DATATYPE=IMSG
```

- Writing a command response, not a message

```
HCPCONSL WRITE,
      DATATYPE=CMNDRESP
```

3. Writing data that you built, or writing data from a message repository

- Writing data that you built

```
HCPCONSL WRITE,
      DATA='--text--'
```

Writing such data does lack the flexibility of using a message repository, but this is a quick and easy method of getting a message displayed.

- Writing data that you built

```
HCPCONSL WRITE,
      DATA=(where,length)
```

Writing such data still lacks the flexibility of using a message repository.

The value for "where" may be a label of a storage location where the data starts ("label") or a register, enclosed in parentheses, that contains the address of the storage location where the data starts ("(Rx)").

The value for "length" (the length of the data to display, in bytes) may be a self-defining term ("8", "X'1F'"), an absolute value ("END-START"), or a register, enclosed in parentheses, that contains the length of the data ("(Rx)").

```
HPCONSL WRITE,  
DATA=(LONGMSG,(R3))
```

- Writing data from a message repository

```
HPCONSL WRITE,  
REPOSNUM='MSmmmmvv'
```

For this format, the macro expansion will use the message number value that has been equated to the label "MSmmmmvv". It is suggested that this label and its value match so as to avoid confusion.

```
MS123401 EQU X'00123401' Good  
MS123401 EQU X'00777706' Bad
```

- Writing data from a message repository

```
HPCONSL WRITE,  
REPOSNUM=label
```

The message number value will be found at the 4-byte field with label "label".

- Writing data from a message repository

```
HPCONSL WRITE,  
REPOSNUM=(Rx)
```

The message number value will be found in register "Rx".

4. Getting back the message number

A standard in CP is that a failed command return a return code, and that the return code match the error message number that explains the error condition. For example, if a command fails and generates the following error message, then that command should end with return code 45:

```
HCP045E $1 not logged on
```

However, the error message value is defined as a hexadecimal value and the return code is defined as a decimal number. You could figure out one from the other, or you can let the HPCONSL macro do it for you by using the RETMSGNUM= keyword.

- Specifying the return code result by label

```
HPCONSL WRITE,  
REPOSNUM=<something>,  
RETMSGNUM=label
```

The message number value will be converted to a return code and stored into the 4-byte field with label "label".

- Specifying the return code result by register

```
HPCONSL WRITE,  
REPOSNUM=<something>,  
RETMSGNUM=(Rx)
```

The message number value will be converted to a return code and stored into register "Rx".

5. Performing substitution with data from a message repository

A message is most useful if it can identify the operand, device, user ID, location, or other data item to which the message refers. Standard messages are frequently built as message patterns so that such identifying detail can be inserted into the message pattern in order to create the final message. The pieces of data to be inserted are known as "substitution text". Substitution text is composed of individual fields each separated by a X'00' byte, all terminated by a X'FF' byte. Each of these fields can be thought of as numbered 1, 2, 3, and so on. These are then substituted into the message pattern where the substitution indicator (character "\$" with a numeric position number, as in "\$1", "\$2", "\$3", and so on) is located.

The location of the substitution text is specified by the SUBDATA= keyword. Notice that a message repository number is required for such substitution.

- Specifying substitution data by label

```
HPCONSL WRITE,
    REPOSNUM=<something>,
    SUBDATA=label
```

The substitution text will be found at the storage location named by label "label".

- Specifying substitution data by register

```
HPCONSL WRITE,
    REPOSNUM=<something>,
    SUBDATA=(Rx)
```

The substitution text will be found at the storage location located by the address in register "Rx".

6. Whose message repository?

You can direct CP to retrieve message patterns from a standard system message repository or from a local repository. This separation is made based on the value of the component ID specified by the HPCONSL macro. Once this decision has been made, CP then selects the repository according to the language ID as established by the SET CPLANGUAGE command.

- IBM's message repository

```
HPCONSL WRITE
    REPOSNUM=<something>,

HPCONSL WRITE,
    COMPID='HCP'
    REPOSNUM=<something>,
```

Both of these formats indicate that the system message repository should be used.

- Getting the message pattern from a local repository

```
HPCONSL WRITE,
    COMPID='CCC'
    REPOSNUM=<something>,
```

The component ID is the 3 characters "CCC".

- Getting the message pattern from a local repository

```
HPCONSL WRITE,
    COMPID=(Rx)
    REPOSNUM=<something>,
```

The 3-character component ID will be found at the storage location located by the address in register "Rx".

7. Sending the data where?

The message generated by HPCONSL can be directed to storage or to any user ID on the system.

- Writing to a GSDBK

```
HCPCONSL WRITE,  
  REPOSNUM=<something>,  
  DESTINATION=GSDBK,  
  RETGSDBKADDR=(Rx)
```

- Writing to the operator

```
HCPCONSL WRITE,  
  DESTINATION='OPERATOR'
```

Use the CP QUERY SYSOPER command to find the current operator *userid*. Use the CP SET SYSOPER command to change the *userid* of the primary system operator.

- Writing to wherever the user wants (terminal, Diagnose code X'8' buffer, IUCV buffer)

```
HCPCONSL WRITE  
  
HCPCONSL WRITE,  
  DESTINATION=*
```

- Writing to the terminal screen

```
HCPCONSL WRITE,  
  DESTINATION=TERMINAL
```

- Writing to another user ID

```
HCPCONSL WRITE,  
  DESTINATION=(Rx)
```

- Writing to another user ID

```
HCPCONSL WRITE,  
  DESTINATION=label
```

Notice that the term 'OPERATOR' is delimited by single quotes ('), but the term TERMINAL is not.

8. Waiting or not waiting?

As you generate a series of messages, you will not know when they reach the user's terminal (it might be in HOLDING, for example). Unless you ask CP to delay your processing, you will continue to execute and will continue to generate messages. If you continue to run and continue to generate messages, the user will not be able to stop the stream of messages. In addition to getting the user upset, you will have wasted system resources generating unwanted messages.

To avoid these problems, your program needs to wait periodically in order for the HCPCONSL macro to return a return code that indicates what is happening with the delivery of the messages. The IOWAIT= keyword provides this ability to wait. If the destination is not the terminal, then you will not wait.

- Waiting for each I/O to complete

```
HCPCONSL WRITE,  
  IOWAIT=IOCOMP
```

This operand indicates that you want to delay until the I/O to the terminal has completed. This will happen for each line. This operand may slow processing significantly if the terminal is connected over a slow network.

- Waiting only when the screen fills

```
HCPCONSL WRITE,  
  IOWAIT=SCREENFULL
```

This operand indicates that you want to delay only when the terminal screen has filled.

- Not waiting for anything

```

HCPCONSL WRITE

HCPCONSL WRITE,
    IOWAIT=NONE

```

This operand, or lack of an operand, indicates that you do not want to delay execution.

9. Detecting if the user wants to you stop

In order to detect that the user wants a stream of messages to stop, the user will typically press PA1. In order to detect this event, your program must have some IOWAIT= specified.

- Placing the return code into a storage field

```

HCPCONSL WRITE,
    IOWAIT=<something>,
    RETCODE=label

```

- Placing the return code into a register

```

HCPCONSL WRITE,
    IOWAIT=<something>,
    RETCODE=(Rx)

```

The meanings of the possible return codes are:

- 00** Success
- 04** Disconnected
- 08** Interrupted by user
- 12** I/O error
- 16** Not logged on
- 20** Soft abend occurred

10. Free flowing or columnar output

Most CP messages are free flowing, there is no need to align columns of data. However, some messages or responses look better if their contents take on a columnar arrangement. The DATAEDIT= keyword directs this decision.

- Free flowing

```

HCPCONSL WRITE

HCPCONSL WRITE,
    DATAEDIT=NORMFORMAT

```

This indicates that the substitution text for the message pattern will cause the message pattern to be shifted as necessary to contain the substitution data.

- Columnar

```

HCPCONSL WRITE,
    DATAEDIT=COLFORMAT

```

This indicates that the substitution text for the message pattern will not cause the message pattern to be shifted. The substitution data will overlay the contents of the message pattern, which area should be blanks in order not to lose any of the data in the message pattern.

Some HCPCONSL Examples

To write a complete HCPCONSL macro, simply review the components discussed above and then combine the operands for the desired attributes. Example one includes the item numbers from the discussion above that will be combined to make the final HCPCONSL macro.

1. Write a command response from the message repository, substituting data located at SAVEWRK2; wait if the screen fills; return the return code to R15 so that it can be checked by a branch table generated by HCPCASRC.

HCPCONSL	WRITE,	Write a command response	X
	DATATYPE=CMNDRESP,		X
	REPOSNUM='MS741701',		X
	SUBDATA=SAVEWRK2,		X
	IOWAIT=SCREENFULL,		X
	RETCODE=(R15)		
HCPCASRC	(RC00,	00: success	X
	RC04,	04: disconnected	X
	RC08,	08: interrupted by user	X
	RC12,	12: I/O error	X
	RC16,	16: not logged on	X
	RC20),	20: soft abend occurred	X
	ELSE=RCXX	?	

2. Write an error message from local message repository whose component ID is "ABC", with the error message in R4 so that we do not confuse anybody reading the code into thinking IBM's message MS009904 is being issued, substituting data located by register R3; do not wait; return the message number into SAVER2 so that the message number becomes the return code from the command.

L	R4,=X'00009904'		
HCPCONSL	WRITE,	Write local error message	X
	COMPID='ABC',		X
	DATATYPE=EMSG,		X
	REPOSNUM=(R4),		X
	RETMSGNUM=SAVER2,		X
	SUBDATA=(R3)		

3. After locating the VMDBK for a specific user ID, write an error message from local message repository whose component ID was passed in R8 at entry to the user ID whose address was passed in R10, with the error message in R4 so that we do not confuse anybody reading the code into thinking IBM's message MS987601 is being issued, substituting data located by register R3; wait until the I/O completes; ignore the message number returned by the macro; return the return code to R15 so that it can be checked by a branch table generated by HCPCASRC.

LA	R0,8	Length of uid	
L	R1,SAVER10	Address of uid	
HCPCALL	HCPCVMD	Find its VMDBK	
BNZ	NOPE	..sorry, not here	
LR	R10,R1	Set R10 -> VMDBK	
L	R4,=X'00987601'	Set R4 = message number	
L	R6,SAVER8	Set R6 -> component ID	
HCPCONSL	WRITE,	Write local error message	X
	COMPID=(R6),		X
	DATATYPE=EMSG,		X
	DESTINATION=(R10),		X
	IOWAIT=IOCOMP,		X
	REPOSNUM=(R4),		X
	RETCODE=(R15),		X
	SUBDATA=(R3)		
HCPCASRC	(RC00,	00: success	X
	RC04,	04: disconnected	X
	RC08,	08: interrupted by user	X
	RC12,	12: I/O error	X
	RC16,	16: not logged on	X
	RC20),	20: soft abend occurred	X
	ELSE=RCXX	?	

Using Other Dynamically Loaded Routines

Other dynamically loaded routines can be called directly by using the HCPCALL macro. But this can be done only if the address constant for the called routine can be resolved by CPXLOAD, so that HCPCALL has its address. CPXLOAD can resolve the address constant only if the proper use of CPXLOAD operands was made.

You must remember that a dynamically loaded routine may be unloaded by CPXUNLOAD if it had been loaded originally with the CPXLOAD TEMPORARY operand.

The following table shows if CPXLOAD can resolve the address constant. For this table, we will assume that the address constant is in routine A and that the address constant refers to a location in routine R. In addition, we will refer to the CPXLOAD load ID for routine A as ID(A), and for routine R as ID(R). The CPXLOAD load ID is important here because the same CPXLOAD load ID (mathematically, $ID(A) = ID(R)$) means that the two routines were loaded by the same CPXLOAD operation, possibly selected by INCLUDE directives.

CPXLOAD for A	CPXLOAD for R	ID(A)::ID(R)	Address resolved?
PERMANENT	PERMANENT	Equal	Yes
TEMP	TEMP	Equal	Yes
PERMANENT	PERMANENT	Not equal	Yes
PERMANENT	TEMP	Not equal	No
TEMP	PERMANENT	Not equal	Yes
TEMP	TEMP	Not equal	No

This table can be summarized to this logic:

```
if A and R were loaded together,
    then the address constant will be resolved
else R must be PERMANENT
```

According to the table, certain address constants cannot be resolved, depending on how the two routines were loaded. Actually, CPXLOAD will be rejected if address constant resolution is prohibited.

To avoid this problem, you can use the HCPCALL with its TYPE=INDIRECT operand. With the TYPE=INDIRECT operand, address constants are not generated by HCPCALL for dynamically loaded routines, so the CPXLOAD restrictions expressed by the above table do not apply. The system overhead for using HCPCALL TYPE=INDIRECT is slightly higher than not using TYPE=INDIRECT, but you gain significant flexibility in being able to call other dynamically loaded-routines, regardless of which operands were used for the CPXLOAD request.

You can even use HCPCALL TYPE=INDIRECT to call routines in the CP nucleus. In this case, HCPCALL is smart enough to realize that the called routine is in the nucleus and that an address constant is always accepted and so the additional system overhead of an indirect call can be avoided.

There are other reasons why an address constant to an external symbol might be used besides being used in HCPCALL. One such reason was alluded earlier in the discussion of Addressing Modes: that is, to locate a data item. As shown by the above table, address constants to external symbols in dynamically loaded routines may not always be accepted. Hence, you may need to use an alternative method to determine the address of an external symbol. One such alternative method would be the use of the CP routine HCPCFDSY. This routine will return the address of a external symbol, which can then be used with HCPCALL. You must be ready, however, for the external to vanish by execution of a CPXUNLOAD request. CP will not prevent use of an old and obsolete address. Care must be exercised to ensure that the referenced routine is not unloaded at an inopportune time.

How Should the Routine be Named?

Simply for consistency, all of the following should start with the same three characters, "xyz":

- The module name, "xyzmmm"
- The module's entry points, "xyzmmme"
- The module's external symbols, "xyzmmfff"
- The xyzMDLAT macro that defines the module's attributes
- Any related local message repository, xyzMES REPOS

As shown above, the first six characters for all entry points and external symbols in a module should be the same. This rule facilitates understanding of how modules interact and which modules contain which data.

Because the first three characters, "xyz", could be used for so many different purposes (assembler external symbol, macro name, CMS file name), they should be as simple as possible. This suggests that:

- "x" should be selected from the alphabetics A-Z.
- "y" should be selected from the alphabetics A-Z and the numerics 0-9.
- "z" should be selected from the alphabetics A-Z and the numerics 0-9.

After selecting the three beginning characters, you need to select the next three characters that complete the module name. Again, for simplicity and to assure general acceptance by all components of z/VM, these three characters should be selected from the alphabetics A-Z and the numerics 0-9.

At this point, all six characters of the module name have been selected. Now you need to select the final two characters for each external entry point or other external symbol in the module. These two characters, like all of the others, should be selected from the alphabetics A-Z and the numerics 0-9.

How Should the Linkage Attributes of the Routine be Specified?

Linkage attributes for dynamically loaded routines are specified in three places:

1. In the xxxMDLAT macro for the routine

The xxxMDLAT macro that you write to describe the linkage for your routines has one MDLATENT statement for each module. The attributes for the entry point names in the module are generally all the same. Because they will be loaded by CPXLOAD, the attribute DYN (indicating that a dynamic SAVBK will be allocated for each entry point when it is called) is required. A choice must be made between attributes MP (indicates that the entry points are multiprocessor capable) and NONMP (indicates that the entry points are only single-processor, or master-processor, capable). Whichever choice is made, that choice must be reflected in the CPXLOAD operand or in a CPXLOAD OPTIONS directive.

```
MDLATENT xxxmod,                                X
          MODATTR=(MP,DYN),                      X
          CPXLOAD=YES
MDLATENT xxxmod,                                X
          MODATTR=(NONMP,DYN),                   X
          CPXLOAD=YES
```

2. By the operands on CPXLOAD:

```
CPXLOAD xxxmod TEXT fm MP
CPXLOAD xxxmod TEXT fm NONMP -or-
CPXLOAD xxxmod TEXT fm NONMP
```

3. By the operands on CPXLOAD OPTIONS directives statement:

```
OPTIONS MP
OPTIONS NONMP -or-
OPTIONS NONMP
```

It is imperative that the linkage attributes specified by these methods agree. If they do not, then the instructions generated by HCPCALL for one type of linkage will not match the attributes for the other type of linkage. Results are unpredictable.

It is strongly suggested that MP always be used and that NONMP be avoided. The NONMP attribute can provide a small degree of serialization because all modules defined as NONMP must run on the same processor, and only one can be running at a time. But, if too many modules must run on the master processor, system performance will suffer. A better choice would be to devise a locking mechanism using the locking services of the HCPLCK module. Then, the modules can run on other processors, freeing the master processor for other work.

The HCPXSERV macro can be used to provide this kind of serialization. See the description of HCPXSERV in Appendix F, “CP Exit Macros,” on page 169.

How are Addresses Resolved?

As the TEXT file (it may have a different file type, depending on how VMSES/E names things) is being read, its information regarding address constants is remembered. This information consists of:

- The name of the control section (CSECT) that contains the address constant
- The location of the address constant in the CSECT
- The length of the address constant
- The name of the external symbol that is referred to by the address constant

This discussion refers both to the SYSGEN process and to the CPXLOAD process, except where their differences are pointed out. This is a discussion of the concept of address constant resolution, and should not be taken as a discussion of programming flow. When the CSECTs are known, the saved information of address constants is then processed. For each address constant, a search is performed to find the external symbol to which the address constant refers and what is the address of that external symbol. The address constant is then adjusted to contain the address of the location to which it refers.

After they have performed all possible address resolution, SYSGEN and CPXLOAD act differently:

- SYSGEN displays a message to the terminal indicating that an address constant could not be resolved, and then SYSGEN discards the address constant information.
- CPXLOAD remembers the address constant information in anticipation of a future CPXLOAD operation defining the external symbol that was not found this time.

Because SYSGEN discards address constant information, address constants in the nucleus cannot be resolved or completed by a later CPXLOAD. Because there is no saved address constant information, there is no way to know that an external symbol being loaded by CPXLOAD could satisfy an address constant in the nucleus.

Can I Add My Own CP IUCV System Service?

CP IUCV system services are things like *MSG, *MSGALL, *SPL, *RPI, *CONFIG. To create an IUCV communication path with a CP system service, use the IUCV Directory Control Statement described in *z/VM: CP Planning and Administration*. For information about the standard set of CP system services, see *z/VM: CP Programming Services*.

CP Exits does not provide for the definition of new CP IUCV system services. Such an extension to the standard set of CP IUCV system services will still require a source modification to all affected CP modules.

What Does the CPXUNLOAD ASYNCHRONOUS Operand Mean?

Normally, CP will process any CP command under the control of the user's Virtual Machine Definition Block (VMDBK). While processing a command, no other activity for that VMDBK is allowed. This means that any other command that the user may issue is held waiting until all prior commands issued by the user have completed.

In a busy system, a CPXUNLOAD command may take longer than the user is willing to wait. To complete a CPXUNLOAD request, CP must prevent all new attempts to use the module, and then wait until all present uses complete. Then, finally, CP can remove the module from storage.

To avoid such a delay, the user may use the CPXUNLOAD ASYNCHRONOUS operand. This operand tells CP to process the command on the system's VMDBK, not on the user's VMDBK. By switching the command to the system's VMDBK, CP can make the user's VMDBK immediately available for more commands. The user at the terminal is not delayed from doing more work.

If the CPXUNLOAD ASYNCHRONOUS operand is used by an EXEC or by a program, this also means that the EXEC or program continues executing beyond the CPXUNLOAD command and may even finish. If the CPXUNLOAD processing generates an informational message or an error message, there is no program ready to receive it. Therefore, all messages will be sent to the user's terminal. This also means that the EXEC or program will not know if the CPXUNLOAD command succeeded or failed.

The best use of the SYNCHRONOUS and ASYNCHRONOUS operands is:

- If the user is issuing the command at the terminal, use (or allow to default to) the ASYNCHRONOUS operand. The user may continue working while the CPXUNLOAD command continues.
- If the user is issuing the command by an EXEC or a program, use the SYNCHRONOUS operand. The EXEC or program can then look for messages or error return code from the command and make decisions about how to proceed.

What Does the CPXLOAD CONTROL Operand Imply?

Generally, you would use the CPXLOAD NOCONTROL operand. However, situations may be such that additional processing is necessary to prepare a proper environment for the loaded routine, or to ensure safe conditions for a later CPXUNLOAD request. This is the intent behind the use of the CONTROL operand: to specify a routine that will provide these extra services.

The CONTROL routine is responsible for ensuring that whatever environment is required for the loaded routine, in fact, exists. For example, the loaded routine may expect that certain control blocks are allocated. The CONTROL routine can perform these allocations in anticipation of the loaded routine needing them. Then later, for a complementary CPXUNLOAD request, the control routine could cleanup and deallocate the control blocks.

The CONTROL routine need not be loaded as part of the same CPXLOAD request. But if not, it must be part of the CP nucleus or have been loaded with the CPXLOAD PERMANENT operand.

Remember that the CONTROL routine must be suitable for use by CPXLOAD and by CPXUNLOAD. It must accept parameters and addresses as passed in registers, and must return with a proper return code in R15. Only routines specifically written to be CONTROL routines should be named by the CONTROL operand. For more details, see [“Additional Information about Control Entry Points” on page 95 in Appendix A, “IBM-Defined CP Exit Points,” on page 89.](#)

What Happens if I Make a Programming Error?

All exit routines, command routines, and Diagnose code routines run as subroutines of CP. No additional "safety nets" are put in place for this environment. This means that any programming errors, for example a protection exception program check, will be treated just as if it happened in IBM CP code. The result will be a CP abend.

It is the severity of such a result that dictates that all exit routines, command routines, and diagnose code routines be fully tested before they are used in a production environment.

Chapter 4. Loading Dynamically into the System Execution Space

This section describes how to load your CP routines into the dynamic area of the system execution space (SXS). You can load your CP routines (or modules) either of the following ways:

- During system initialization, using the CPXLOAD configuration file statement.
- During system operation, using the CPXLOAD command. (You can also use the CPXLOAD command to reload a module that was unloaded using a CPXUNLOAD command.)

The decision to use CPXLOAD during or after initialization depends on:

- When you will need to use the routine or module
- How convenient it is for you to package the files

Understanding CPXLOAD

To dynamically load your CP routines into the system execution space, you must use either the CPXLOAD command or configuration file statement. CPXLOAD has a wide variety of operands that you can use to tell CP the characteristics of the routines or modules that you are loading. These operands are explained in the following sections.

Using the DELAY and NODELAY Operands

The DELAY and NODELAY operands tell CP when to load your CP routines during initialization. These operands have no meaning after initialization completes, but are included on the CPXLOAD command for the sake of consistency. If you specify them on a CPXLOAD command, CP ignores them.

If you specify the DELAY operand during system initialization, CP waits until after all CPACCESS statements are processed before it processes the CPXLOAD statement. This allows you to put your CP routines on any disk to which CP will have access. DELAY is the default.

If you specify the NODELAY operand during system initialization, CP processes the CPXLOAD statement immediately. In this case, the CPXLOAD statement is only guaranteed to complete successfully if you put your CP routines on the parm disk. Until CP processes a CPACCESS statement, the parm disk is the only disk that CP can access.

If you are not sure whether to use DELAY or NODELAY, ask yourself the following questions:

- Do you need to use your CP routine or module during initialization? For example, suppose you are loading a module that you designed to be used during the parsing of the system configuration file (this would be true for external symbols specified by the EXTERNAL_SYNTAX statement).

If the answer is "yes", use NODELAY.

- Did you place your CP routine or module on a CMS minidisk other than the parm disk?

If the answer is "yes", use DELAY.

Using the PERMANENT and TEMPORARY Operands

The PERMANENT and TEMPORARY operands tell CP whether your CP routines or modules can be unloaded with a CPXUNLOAD command.

If you specify the PERMANENT operand, CP will not allow the files to be unloaded. This, typically, would be for a module that is fully-debugged, fully-operational, and required at all times. For example, security-related modules on production systems would probably be loaded as permanent and dynamically loaded modules on test systems would probably be loaded as temporary (in preparation for a new version or fix).

Another issue is that of address constants pointing between modules. If two different modules (Q and R) are loaded by two different CPXLOAD operations, and module Q contains an address constant that refers to module R, then module R must have been loaded with CPXLOAD PERMANENT. If both modules Q and R must be loaded separately with CPXLOAD TEMPORARY operand, then one or the other module must be redesigned to eliminate the address constant. If both modules Q and R can be loaded together with CPXLOAD TEMPORARY operand, then neither need be redesigned, the address constant can be resolved.

Using the CONTROL and NOCONTROL Operands

Usually, no special processing is needed during loading or during unloading, so CPXLOAD NOCONTROL is appropriate. If special processing is needed before CPXLOAD can be declared successful, or before CPXUNLOAD may commence, CPXLOAD CONTROL could be used. The control routine must be "aware" of the CPXLOAD and the CPXUNLOAD environments, or disaster will result. Being "aware" means that the routine must be ready for the data and addresses passed in registers from the CPXLOAD or CPXUNLOAD operation.

An example of a design when CONTROL would be appropriate might be this:

Suppose that the module being loaded was written to require the existence of its Component ID Block (CMPBK) as well as certain data areas already allocated and located by pointers in the CMPBK. During system operation when the module would be called, this structure of control blocks must already be in place, even for the very first time that the module is called. In this case, a control routine could also be written so that when it is called during CPXLOAD processing, this structure of control blocks could be created. Also, if a CPXUNLOAD command were issued, the control routine would get control and could "tear down" this structure of control blocks.

CONTROL gives more control over CPXLOAD and CPXUNLOAD, but must be used correctly. Indiscriminate, ill-advised use of CONTROL can cause problems. For more details, see ["Additional Information about Control Entry Points"](#) on page 95 in Appendix A, ["IBM-Defined CP Exit Points,"](#) on page 89.

Using the MP, NOMP, and NONMP Operands

Good programming practice calls for all routines in a multiprocessor system to be multiprocessor capable (MP). Therefore, avoid writing routines that are only for the master processor (NOMP or NONMP) unless a specific routine cannot be designed to be multiprocessor capable; in such a case the routine must be indicated as CPXLOAD NOMP.

Using the LET and NOLET Operands

When a file is being loaded dynamically, the first character in each record is significant in that it indicates the type of record processing that is required.

- An asterisk (*) in column 1 indicates that the record is a comment and should be ignored.
- The value X'02' indicates that the record is a TEXT record, which CPXLOAD will process.
- A blank in column 1 denotes a possible CPXLOAD directive, which will be validated and handled.
- Anything else in column 1 would be considered an invalid record and would typically be flagged as an error.

There are some cases where finding an invalid record based on the value in column 1 is not actually a problem. For example, an assembler utility, such as VMFHASM or VMFHLASM, will often leave information in the TEXT file that is meant to be ignored by a loader program. For example, here are such records from a TEXT file for HCPPTU where the VMFHASM EXEC has included a record (the first record) that does not conform to the rules mentioned above.

```
G54306HP PUT UM23148 SYSTEM HUNG, ABENDMCW002 OR ABENDPAG001.      N
*      HCPPTU  G54306HP H1 FCP12H  10/22/92  15:39:00
* PREREQ: NONE
* CO-REQ: NONE
* IF-REQ: NONE
```

You can control the type of checking that is done on the records in the file that CPXLOAD processes by using LET and NOLET.

- LET, which is the default, tells CP to load the specified file and to ignore records that are either completely blank or that contain an unexpected value in the first column. This is meant to accommodate the non-comment information that can be left in a TEXT file by an assembler utility, such as VMFHASH or VMFHLASM. It saves you the added work of editing your TEXT file to delete these extraneous non-comment lines.
- NOLET tells CP to stop loading the file when an invalid record is detected. You would use NOLET in cases where you have removed all of the extraneous information from your text file, and therefore expect all the records to be valid.

In the example above,

- LET will allow CPXLOAD to succeed,
- NOLET will cause CPXLOAD to fail, because of record 1.

How to Package Modules

How modules are written is a design decision.

- One large module, written as a single CSECT, containing multiple entry points, or
- Multiple modules, each written as a single CSECT, each containing a single entry point.

Storage for dynamically loaded modules is performed so that each CSECT is loaded into storage in its own page. A module written as a single CSECT with multiple entry points will use less storage than multiple modules each written as a single CSECT each with a single entry point. But, if one entry point evolves and the CSECT exceeds 4 KB in size, difficult rework of the module may be necessary in order to find another base register or in order to split the CSECT into multiple separate CSECTs.

Once the arrangement of CSECTs has been determined, the decision must be made about how to package their TEXT files produced by the IBM High Level Assembler.

- Should they remain separate files loaded by separate CPXLOAD requests?
- Should they remain separate files loaded by a single CPXLOAD request using INCLUDE directives?
- Should they be packaged into a single TXTLIB file but still loaded by separate CPXLOAD requests?
- Should they be packaged into a single TXTLIB file and loaded by a single CPXLOAD request using INCLUDE directives?
- Should they be packaged into a single TXTLIB file and loaded by a single CPXLOAD request using the pattern matching capabilities of the MEMBERS operand?

Each of these alternatives affect ease of loading and ease of distribution. There is no single packaging scheme that is right for all situations.

Examples of CPXLOAD Commands

Now that we have described what the operands on CPXLOAD mean and have given you information about packaging modules, suppose we take you through some scenarios so that you can actually use this information.

A Note about the Examples

The examples in this section show you lists of instructions. These instructions show you how to accomplish a task using the basic steps and the basic order in which those steps should be accomplished. However, each situation is different and we cannot begin to document all of the possible variations. So, use the steps in these examples as guidelines, not hard-and-fast rules.

CPXLOAD Example 1: Installing a New CP Command

Suppose that you want to install a new CP command routine for users with privilege class G authority. For this example, complete the following steps in the following order:

1. Decide the name of the new CP command. (We will use *cmd* to represent the command name that you chose.)
2. Decide on the name of the new source module. (We will use *fn* to represent the module name that you chose.)
3. Decide on the two additional characters to add to the module name (*fn*) to generate the entry point name of the command processor. (We will use *ep* to represent the entry point name that you chose.)
4. Write the new module.
5. Assemble the module with the IBM High Level Assembler.
6. Copy the resulting TEXT file to a minidisk that will be (or is) accessed by CP.
7. Make the minidisk accessible to CP using the CPACCESS command.
8. Define the command using the DEFINE COMMAND command.

```
define command cmd enable privclass g ibmclass g epname ep
```

9. Decide which CPXLOAD attributes to use:

- You wrote the routine to be multiprocessor capable, so use MP, the default.
- You wrote the routine such that it needs no special set up or clean up processing, so use NOCONTROL.
- You want to be able to load a new version of the routine, if this one needs to be replaced, so use TEMPORARY.

10. Load the TEXT file using the CPXLOAD command:

```
cpxload fn text temp nocontrol
```

After completing the above steps, you will have made the new CP command available to your users.

CPXLOAD Example 2: Installing a New Diagnose Code

Suppose that you want to install a new Diagnose code routine for users with privilege class G authority. For this example, complete the following steps in the following order:

1. Decide on the number for the new Diagnose code. (We will use *diag* to represent the Diagnose code that you chose.)
2. Decide on the name of the new source module. (We will use *fn* to represent the module name that you chose.)
3. Decide on the two additional characters to add to the module name (*fn*) to generate the entry point name of the Diagnose code processor. (We will use *ep* to represent the entry point name that you chose.)
4. Decide on the two additional characters to add to the module name (*fn*) to generate the entry point name that CP will call when the module is loaded. (We will use *ct* to represent the entry point name that you chose.)
5. Write the new module.
6. Assemble the module with the IBM High Level Assembler.
7. Copy the resulting TEXT file to a minidisk that will be (or is) accessed by CP.
8. Make the minidisk accessible to CP using the CPACCESS command.
9. Define the command using the DEFINE COMMAND command.

```
define diagnose diag enable privclass g epname ep
```

10. Decide which CPXLOAD attributes to use:

- You wrote the routine to be multiprocessor capable, so use MP, the default.
- You wrote the routine such that it needs special set up processing, so use CONTROL.
- You want this routine to be available forever and never to be unavailable. Operation of your VM service is dependant on this Diagnose code being available, so use PERMANENT.

11. Load the TEXT file with the CPXLOAD command:

```
cpxload fn text perm control ct
```

After completing the above steps, you will have made the new Diagnose code available to your users.

CPXLOAD Example 3: Installing a New Message

Suppose that you want to install a CP message that replaces a standard CP message. For this example, complete the following steps in the following order:

1. Decide which message (or messages) you will be changing.
2. Decide on the name of the new source module. (We will use *repos* to represent the source name that you chose for your message repository.)
3. Write the new message repository.
4. Assemble the module with the CMS GENMSG command.
5. Copy the resulting TEXT file to a minidisk that will be (or is) accessed by CP.
6. Make the minidisk accessible to CP using the CPACCESS command.
7. Decide which CPXLOAD attributes to use:
 - Message repositories contain only data, so the MP or NONMP decision is immaterial. It is simplest to use MP, the default.
 - Message repositories need no special set up or clean up processing, so use NOCONTROL.
 - You want to be able to load a new version, if this one needs to be replaced, so TEMPORARY should be used.
8. Load the TEXT file with the CPXLOAD command.

```
CPXLOAD repos TEXT TEMP NOCONTROL
```

9. Associate the message repository with component ID HCP, the CP component ID, by using the ASSOCIATE MESSAGE command.

```
ASSOC MESSAGES COMP HCP EPNAME repos
```

Having completed the above steps, you will have caused CP to use your new message rather than the standard message.

Examples of CPXLOAD Directives

Suppose that you have written a number of modules that, taken together, comprise the routines for a user Diagnose code. Suppose that these routines are named like this, with attributes as indicated.

- DGA1FCEP

The initial entry for processing of your Diagnose code. This entry point will examine the subcode parameter and route further execution as appropriate. This routine is multiprocessor capable.

- DGB1FC00

This is the routine that processes subcode 00. This routine is multiprocessor capable.

- DGC1FC04

This is the routine that processes subcode 04. This routine is not multiprocessor capable and therefore must run only on the master processor.

- DGD1FC08

This is the routine that processes subcode 08. This routine is multiprocessor capable.

For the convenience of your customers (by the way, you are a vendor of this code), you have packaged these routines into members of a single TXTLIB file, named DGN1FC TXTLIB. As another convenience to your customers, you have included member RULES in DGN1FC TXTLIB to contain CPXLOAD directives so that your customers need not be concerned with such details, and cannot specify them erroneously.

This, then, would be the CPXLOAD command that would cause all of your routines to be loaded.

```
CPXLOAD DGN1FC TXTLIB MEMBER RULES
```

The RULES member of DGN1FC TXTLIB is shown below.

```
RULES TEXT A1 F 80 Trunc=80 Size=9 Line=0 Col=1 Alt=0

|...+....1....+....2....+....3....+....4....+....5....+..
0 *** Top of File ***
1 OPTIONS TEMPORARY NOCONTROL
2 OPTIONS MP
3 INCLUDE DGN1FC TXTLIB MEMBER DGA1FCEP
4 OPTIONS MP
5 INCLUDE DGN1FC TXTLIB MEMBER DGB1FC00
6 OPTIONS NONMP
7 INCLUDE DGN1FC TXTLIB MEMBER DGC1FC04
8 OPTIONS MP
9 INCLUDE DGN1FC TXTLIB MEMBER DGD1FC08
10 *** End of File ***
```

Figure 3. RULES member of DGN1FC TXTLIB

The operands specified on each OPTIONS directive match the descriptions above. The order of the statements in RULES is significant. The operands on each OPTIONS directive applies to the the CSECTs that are it. Operands on subsequent OPTIONS directives override operands of previous OPTIONS directives.

Here is a description of contents of the RULES member:

Line

Contents

1

This OPTIONS directive specifies that the dynamically loaded routines can be unloaded in the future with a CPXUNLOAD command. Also, there is no control entry point associated with these dynamically loaded routines.

2-3

The routine DGA1FCEP is multiprocessor capable.

4-5

The routine DGB1FC00 is multiprocessor capable.

6-7

The routine DGC1FC04 is not multiprocessor capable.

8-9

The routine DGD1FC08 is multiprocessor capable.

For a full description of CPXLOAD directives and especially the OPTIONS directive, refer to [Appendix C, "CPXLOAD Directives,"](#) on page 151.

Chapter 5. Controlling a Dynamically Loaded Routine

This section shows how you can control your dynamically loaded routines. How you control a dynamically loaded routine depends on the reason that you loaded it: to be a CP command routine, to be a Diagnose code routine, to be a CP exit routine, or to be a message repository.

Note: Once defined, commands, subcommands, aliases, and Diagnose codes cannot be deleted. They may be altered in various appropriate ways, but they remain in existence until a SHUTDOWN or RESTART IPL is done.

Controlling a Dynamically Loaded CP Command Routine

You can use any of the following commands and configuration file statements to control your dynamically loaded CP command routine:

DISABLE COMMAND or CMD command or statement

Use DISABLE to make a CP command or subcommand or alias unavailable. DISABLE can be undone by ENABLE.

ENABLE COMMAND or CMD command or statement

Use ENABLE to resume the usability of a CP command, subcommand, or alias. By default, a new command or alias is defined initially as disabled. ENABLE may be needed to make the command, subcommand, or alias usable.

CPXLOAD command or statement

Use CPXLOAD to load one or more modules into the system execution space. These files must be on a CMS minidisk accessed by CP.

CPXUNLOAD command

Use CPXUNLOAD to remove one or more modules that were dynamically loaded into the system execution space by a previous CPXLOAD request. Only modules that were loaded with the TEMPORARY operand can be unloaded.

DEFINE COMMAND or CMD command or statement

Use DEFINE COMMAND to define a new CP command, subcommand, or a new version of an existing CP command or subcommand.

DEFINE ALIAS command or statement

Use DEFINE ALIAS to define a new alias command for an existing CP command or subcommand. The existing CP command is termed the base command, because the alias command points to it for its attributes. An alias command cannot be defined for another alias command. Once defined, an alias name cannot be used again. An alias command cannot be modified or eliminated; it can only be disabled. Use QUERY CPCMDS to see which commands are alias commands and which are base commands.

MODIFY COMMAND or CMD command or statement

Use MODIFY COMMAND to change certain attributes of any CP command or subcommand. (An alias command cannot be modified.) Among the attributes that can be changed by MODIFY are the command processing routine and privilege classes. A user whose privilege classes do not overlap the command's privilege classes will not be able to issue the command.

QUERY CPCMDS command

Use QUERY CPCMDS to display attributes of any CP command or subcommand. Among the data displayed will be the status of the command (enabled or disabled), the command processing routine, and the privilege classes. A disabled base command or alias command cannot be issued. The command must be enabled before it can be used.

LOCATE SYMBOL command

Use LOCATE to display the address of any external symbol. When the address is known, you can use the DISPLAY (Host Storage) command to examine the module.

Controlling a Dynamically Loaded Diagnose Code Routine

You can use any of the following commands and configuration file statements to control your dynamically loaded Diagnose code routine:

DISABLE DIAGNOSE command or statement

Use DISABLE to make a Diagnose code routine unavailable. DISABLE can be undone by ENABLE.

ENABLE DIAGNOSE command or statement

Use ENABLE to resume the usability of a Diagnose code routine. By default, a new Diagnose code is defined initially as disabled. ENABLE may be needed to make the Diagnose code usable.

CPXLOAD command or statement

Use CPXLOAD to load one or more modules into the system execution space. These files must be on a CMS minidisk accessed by CP.

CPXUNLOAD command

Use CPXUNLOAD to remove one or more modules that were loaded into the system execution space by a previous CPXLOAD request. Only modules that were loaded with the TEMPORARY operand can be unloaded.

DEFINE DIAGNOSE command or statement

Use DEFINE DIAGNOSE to define a new Diagnose code.

MODIFY DIAGNOSE command or statement

Use MODIFY DIAGNOSE to change certain attributes of any Diagnose code (except Diagnose code X'214'). Among the attributes that can be changed by MODIFY DIAGNOSE are the Diagnose code processing routine and privilege classes. A user whose privilege classes do not overlap the Diagnose code's privilege classes will not be able to issue the Diagnose code.

QUERY DIAGNOSE command

Use QUERY DIAGNOSE to display attributes of any Diagnose code. Among the data displayed will be the status of the Diagnose code (enabled or disabled), the Diagnose code processing routine, and privilege classes. A disabled Diagnose code cannot be issued. The Diagnose code must be enabled before it can be used.

LOCATE SYMBOL command

Use LOCATE to display the address of any external symbol. When the address is known, you can use the DISPLAY (Host Storage) command to examine the module.

Controlling a Dynamically Loaded CP Exit Routine

You can use any of the following commands and configuration file statements to control your dynamically loaded CP exit routine:

DISABLE EXITS command or statement

Use DISABLE to tell CP not to call any routines associated with the specified CP exit number. DISABLE can be undone by ENABLE.

ENABLE EXITS command or statement

Use ENABLE to tell CP to resume calling routines associated with the specified CP exit number. By default, ASSOCIATE EXIT is defined initially as disabled. ENABLE may be needed to tell CP to start calling the routines associated with the specified CP exit number.

CPXLOAD command or statement

Use CPXLOAD to load one or more modules into the system execution space. These files must be on a CMS minidisk accessed by CP.

CPXUNLOAD command

Use CPXUNLOAD to remove one or more modules that were loaded into the system execution space by a previous CPXLOAD request. Only modules that were loaded with the TEMPORARY operand can be unloaded.

DEFINE EXIT command or statement

Use DEFINE EXIT to define an exit point dynamically. You place the exit point by specifying the module name and the offset within the module where the exit is to reside, as well as the instruction at

that offset. The exit is taken before that instruction is executed. You may also specify the parameters that are to be provided when the exit is called.

MODIFY EXIT command or statement

Use MODIFY EXIT to modify or remove a dynamic exit point defined by the DEFINE EXIT command or statement. You can change the exit location, characteristics, and parameters.

ASSOCIATE EXIT command or statement

Use ASSOCIATE EXIT to associate a list of entry points with a CP exit number. If the CP exit point is reached, then CP tries to call the entry points if the CP exit point is enabled.

DISASSOCIATE EXIT

Use DISASSOCIATE EXIT to delete the list of entry points from the exit number. The exit remains enabled or disabled, whichever it was. If enabled, CP will continue to perform exit call initialization. If enabled and no entry points are associated with the exit then the CP exit call initialization is unnecessary processing. To avoid this overhead, issue DISABLE EXIT for the exit number.

QUERY EXIT

Use QUERY EXIT to display attributes of any exit number. Among the data displayed will be the status of the exit (enabled or disabled), the exit processing routine, statistics, and so on. For a dynamic exit, the exit location, characteristics, and parameter definitions will also be displayed.

LOCATE SYMBOL

Use LOCATE to display the address of any external symbol. When the address is known, you can use the DISPLAY (Host Storage) command to examine the module.

Controlling a Dynamically Loaded Local Message Repository

You can use any of the following commands and configuration file statements to control your dynamically loaded local CP message repository:

CPXLOAD command or statement

Use CPXLOAD to load one or more modules into the system execution space. These files must be on a CMS minidisk accessed by CP.

CPXUNLOAD command

Use CPXUNLOAD to remove one or more modules that were loaded into the system execution space by a previous CPXLOAD request. Only modules that were loaded with the TEMPORARY operand can be unloaded. Modules still in use (still associated as message repositories) cannot be unloaded until disassociated.

ASSOCIATE MESSAGES or MSGS command or statement

Use ASSOCIATE MESSAGES to associate one or more entry point names with a message repository component ID. This will enable your modules to write your messages.

DISASSOCIATE MESSAGES

Use DISASSOCIATE MESSAGES to undo ASSOCIATE MESSAGES requests.

QUERY CPLANGLIST ASSOCIATED

Use QUERY CPLANGLIST ASSOCIATED to display message repository component IDs and their associated entry point names. You can use the component ID and language ID displayed in DISASSOCIATE MESSAGES commands to break or remove the association, thereby allowing CPXUNLOAD to be used.

LOCATE SYMBOL

Use LOCATE SYMBOL to display the address of any external symbol. When the address is known, you can use the DISPLAY (Host Storage) command to examine the module.

IPL Parameter NOEXITS

If you find yourself in a situation where a CP exit routine is causing your system to not initialize, there is help. This help is in the form of the new IPL parameter NOEXITS. Like all other IPL parameters, the NOEXITS parameter would be supplied by you on the Stand Alone Programmer Loader initial panel.

If specified, the NOEXITS IPL parameter will cause the following to happen:

Controlling Routines

- CPXLOAD statement will be discarded.
- EXTERNAL_SYNTAX statement will be discarded.
- ASSOCIATE EXIT statement operand ENABLE will be ignored.
- ASSOCIATE MESSAGES statement will be discarded.
- ENABLE EXITS statement will be discarded.

All of these statements will be parsed for syntactical correctness, but will not be otherwise processed.

The intended effects of NOEXITS are that only modules in the nucleus be used, and that nothing dynamically "attached" to the nucleus be used or called.

Chapter 6. Defining and Modifying Commands and Diagnose Codes

This section provides an overview of the basic information that you will need to define your own command or Diagnose code, or modify an existing one. Note that we cannot possibly discuss all topics relating to coding CP functions. In this section, we discuss:

- A high level overview of the command and Diagnose code control blocks so that you understand what the DEFINE COMMAND (and DEFINE DIAGNOSE) and MODIFY COMMAND (and MODIFY DIAGNOSE) commands are doing.
- How to code a command or Diagnose code. This includes inputs on entry from the router and expected outputs from your handler.
- External security manager (ESM) considerations.
- The commands necessary to define or modify a command or Diagnose code.
- How to disable the command or Diagnose code.

CMDBKs, DGNBKs and You

In order to understand how to code your command and Diagnose code, and what operands to specify on the DEFINE and MODIFY commands, you should understand where CP stores the information it uses for command and Diagnose code routing. This section describes this structure.

CMDBKs

In CP, all commands, the SET subcommands, the QUERY subcommands and the QUERY VIRTUAL subcommands are processed by the command router. Each one of these entities has a CMDBK control block defined with information concerning the command. These control blocks are chained together in a manner that allows quick efficient searching. There is one chain for commands, another for SET subcommands, and so on. If a command has multiple IBM Classes then there will be a CMDBK for each class.

The CMDBK information includes:

- Command name and its minimum abbreviation
- Command handler name and addressing information
- IBM Class and User Privilege Classes
- Control flags concerning ESM settings and other CP functions
- Space set aside for installation variables (field names begin with the string "CMDUSR")

DGNBKs

Just as each command has a control block with information about it, each Diagnose code has a DGNBK control block. The DGNBKs are accessed in an array by the Diagnose code router. Diagnose codes are numbered from x'000' to x'3FC' and incremented by 4 (such as 0,4,8,C,10, ... 3FC). Numbers x'100' through x'1FC' are reserved for customer and vendor use.

The DGNBK information includes:

- Diagnose code number
- Diagnose code handler name and addressing information
- User Privilege Classes
- Control flags concerning ESM settings and other CP functions

- Space set aside for installation variables (field names begin with the string "DGNUSR")

Relationship between IBM Class and User Privilege Class

A command or Diagnose code may be able to perform different functions depending upon the privilege class of the user that invoked it.

For a Diagnose code, the various functions provided by a Diagnose code are identified by the user privilege class. Diagnose code definitions do not include IBM class.

The various functions provided by a command are identified by the IBM class, which is derived from the user privilege class. The types of authorities for the various IBM classes are shown in [Table 3 on page 44](#).

Table 3. IBM Class Definitions

Class	User and Function
A	System Operator:. The class A user controls the z/VM system. The system operator is responsible for the availability of the z/VM system and its resources. In addition, the system operator controls system accounting, broadcast messages, virtual machine performance options, and other options that affect the overall performance of z/VM.
B	System Resource Operator. The class B user controls all the real resources of the z/VM system, except those controlled by the system operator and the spooling operator.
C	System Programmer. The class C user updates or changes system-wide parameters of the z/VM system.
D	Spooling Operator. The class D user controls spool files and the system's real reader, printer, and punch equipment allocated to spooling use.
E	System Analyst. The class E user examines and saves system operation data in specified z/VM storage areas.
F	Service Representative. The class F user obtains, and examines in detail, data about input and output devices connected to the z/VM system. This privilege class is reserved for IBM use only.
G	General User. The class G user controls functions associated with a particular virtual machine.
0	No IBM class. A command with IBM class 0 is associated with the version of the command with PRIVCLASSANY. A command routine handling multiple functions for a command with some IBM classes and no IBM class would typically look for the IBM class authorities first and then, if no IBM class authority was granted, would handle the case of no IBM class.

When the CP system is shipped, there is usually a one-to-one relationship between the IBM class (A-G, 0) and the user privilege classes (A-Z, 1-6 and 'any') to which it is mapped. However, installations may change this mapping by using system configuration file MODIFY statements and the CP MODIFY command.

Coding Your Command and Diagnose Code

Now that you know how CP identifies the commands and Diagnose codes, we can describe items you need to consider when you code your command or Diagnose code. Once again, we will not cover all possible rules and conventions for writing code in CP. Instead, we will cover the interfaces that relate to invoking your code and returning from your code.

Coding a Command Handler

Input from the Command Router

The command router determines whether the user can issue a particular command. It will invoke the command handler if it determines that the user has the user privilege class necessary to invoke at least one of the versions of a command. The command router makes this determination based on the command or subcommand name. However, some command versions are only differentiated by a specific option that is allowed on the command line. Because a user may be authorized for more than one version of a command, CP requires that all versions of the command have the same entry point. It is up to the command handler to determine which version of the command it should perform.

When control is given to the command handler, a number of registers are initialized with information about the command. The registers contain:

Register

Contents

R0

Length of the command, as specified.

R1

Address of the command in the GSDBK.

R2

Zero

R3

First 4 bytes of the command name from the base CMDBK. Because of command aliasing, this may not be the same command name as indicated by R0 and R1. That is because the command issued may be an alias for the actual command to be invoked. For example, if the user issued MSG that is an alias of MESSAGE then the base command is MESSAGE and this register would contain the string "MESS".

R4

Second 4 bytes of the command name from the base CMDBK. (See the explanation for R3 about base CMDBKs.)

R5

Address of the base CMDBK. This is the CMDBK for one of the versions of the base CMDBK. If the command has multiple versions this address need not be the version that the command handler will select to handle.

R6

Zero

R7

Zero

R11

Address of the VMDBK of the issuer.

R13

Address of the SAVBK to be used during the call.

R14

Linkage register

R15

Linkage register

The contents of the registers that are not specified in the preceding list should be considered undefined. No assumptions should be made on their contents.

Among the information contained in the VMDBK is the address of the command GSDBK. This value is contained in the VMDCFBUF field.

In addition to specifying certain registers with information about the command being invoked, the VMDBK is in CONSOLE FUNCTION MODE. This means that the other virtual processors (VMDBKs) for this user are not actively being dispatched.

Determining the Command Version (IBMCLASS)

On entry from the command router, the command handler knows that the user has been authorized to issue one of the versions of the command. The VMDCTYPE field in the base VMDBK contains the IBMCLASS values for the versions of the command that the user can issue. The byte consists of one flag bit for each IBMCLASS values A-H (see USERCLS0 bit definitions in HCPCLASS COPY for a mapping of the flag values).

No flag bit is set for IBMCLASS 0 because any other version of the command should be considered a superset of the function provided by the IBMCLASS 0 version. Thus, if the user did not issue any of the other versions of a command, the command handler knows that the IBMCLASS 0 version should be handled.

ESM Interactions and Auditing

Many, if not all, ESMs support logging of the commands that a user invokes. This logging provides an audit trail of the commands issued by a user for security analysis. Whether the ESM call for auditing is performed by the command router or command handler depends on the type of command and its versions. Single version commands may allow the command router to perform the auditing call. If there is no "security" difference between the versions of a multi-version command then the command router may perform the auditing call. If it is necessary to differentiate between which version of a command the user invoked then the command handler has to perform the ESM call for auditing.

In addition to auditing, some systems run ESMs at a higher level of security protection that require additional security verification dependent upon the version of the command and its operands. The command handler handles the invocation of the ESM for this level of security verification.

For information on interfacing with ESMs, see the CP Access Control Interface topic in [z/VM: CP Planning and Administration](#).

Exiting to the Command Router

Upon completion of processing, the command handler should return to the command router with register 2 set to zero or the message number of the error message that it generated. This is normally accomplished by storing the value in SAVER2 prior to invoking the HCPEXIT macro. The value returned in SAVER2 is the command completion return code that is returned to CMS and displayed on the CMS READY prompt.

The command router will attempt to process the next command (if one exists).

Coding a Diagnose Code Handler

Input from the Diagnose Code Router

While Diagnose codes that are defined using the DIAGNOSE macro and assembled into HCPHVB allow selection of different types of Diagnose code router invocation, such as HCPCALL or HCPGOTO linkage, the Diagnose codes that we discuss in this section are loaded using the CPXLOAD command and defined using the DEFINE DIAGNOSE command. The Diagnose code router will invoke these Diagnose code handlers using HCPCALL with INDIRECT linkage. Additional information on Diagnose codes and descriptions of the errors reported by the Diagnose codes is contained in [z/VM: CP Programming Services](#).

As with commands, the Diagnose code router verifies that the user has the necessary privilege class to invoke the Diagnose code. The Diagnose code handler may need to handle additional authorization checks. Any ESM calls for additional checks would be contained in the Diagnose code handler.

When control is given to the Diagnose code handler, a number of registers are initialized with information about the Diagnose code. The registers contain:

Register Contents

R5

Address of GUEST 'Rx' register value in the VMDBK.

R6

Address of GUEST 'Ry' register value in the VMDBK.

R8

Address of the DGNBK.

R11

Address of the VMDBK of the virtual processor for the user from which the Diagnose code was issued.

R13

Address of the SAVBK to be used during the call.

Additional information pertaining to the Diagnose code may be obtained using the HCPDCODE macro. This data includes the Diagnose code number whose value will be between X'100' and X'1FC'.

ESM Interactions and Auditing

For information on interfacing with ESMs, see the CP Access Control Interface topic in [z/VM: CP Planning and Administration](#).

Exiting to the Diagnose Code Router

Any return codes or condition codes to be returned to the user are set by the Diagnose code handler. HCPGSVC0, HCPGSVC1, HCPGSVC2, and HCPGSVC3 routines provide function to set virtual condition codes 0, 1, 2 and 3, respectively, in the VMDBK. The return codes are usually set in the Ry+1 register or one of the other Rx, Rx+1, or Ry registers. The address of the fields for the Rx and Ry registers are passed as input to the Diagnose code handler and can be used to update these fields.

In addition, some other conditions (e.g. return a program check) may be passed back to the Diagnose code router to handle using a combination of values set in register 15 and other registers. For more information on the recognized return code values for register 15, see the description of the DEFINE DIAGNOSE command in [z/VM: CP Commands and Utilities Reference](#) concerning the CHECKR15 operand.

Defining your Command and Diagnose Code

After you have coded your command or Diagnose code, you can load the code into the system execution space using the CPXLOAD command or configuration file statement. See Chapter 4, “Loading Dynamically into the System Execution Space,” on page 33. You should note that you can define the command or Diagnose Code before you load the code but we recommend that you load the code first, because this will save you from getting information messages warning you about the absence of the code when you attempt to perform the DEFINE command and specify an EPNAME that is not loaded.

If your system is running an ESM, you should define your commands and Diagnose codes using configuration file statements. This is because the ESM takes a snapshot of the list of commands and Diagnose codes that are part of the system. Also, some ESMs support selective auditing where a copy of the list of commands and Diagnose codes are maintained for a user's virtual machine when they log on. Changing the table after the ESM has acquired its snapshot will cause unpredictable results. By defining any new commands (including new versions of a command) or Diagnose codes using configuration file statements, you limit changes to the command and Diagnose code structure at IPL time prior to the autologging of the ESM virtual machine or other users.

Next, you should consider the operands and options that you will wish to specify on the DEFINE COMMAND and DEFINE DIAGNOSE commands. These commands inform the system about your command and Diagnose code. As the result of processing these commands, the system creates the necessary CMDBK and DGNBK with the control flags and settings set for your code.

There are a number of operands on the DEFINE COMMAND and DEFINE DIAGNOSE instruction that relate to ESM processing. The PROC operand indicates that the command/Diagnose code router should not perform any ESM calls. Instead, the command/Diagnose code handler is expected to perform the necessary ESM calls.

AUDIT indicates that the command/Diagnose code will be audited by the ESM. It is the responsibility of the handler to perform the AUDIT call to the ESM if PROC was specified. The AUDIT setting may be modified by the ESM.

PROT indicates that the command or Diagnose code should be protected by the ESM. When commands or Diagnose codes with PROT are invoked the call to the ESM will inform the ESM that it should search authorization lists to determine whether the user may invoke the command or Diagnose Code. The PROT setting may be modified by the ESM if VPROT was also specified.

MAC indicates that the command should be protected by the ESM at a mandatory access control level. ESMs that support this level of protection perform security label validation. The MAC setting may be modified by the ESM if VMAC was also specified. See your ESM for documentation on the level of protection that it provides.

Defining Your Command

In this section we discuss the DEFINE command operands which are unique to command processing.

Some ESMs restrict the usable characters that you may assign to a command name. For example, some ESMs use the underscore character (_) as an operand separator. You should consult your ESM documentation for information on any restrictions that apply to your system.

Prior to defining your new command you should verify that the command or subcommand name does not conflict with any existing command or subcommand. The QUERY CPCMDs command can be used to verify your command and its abbreviation. For example, to determine if any other QUERY subcommands have a 3 character abbreviation of "ACC", which we wish to use for our new "QUERY ACCUMULATOR" command, you could issue:

```
QUERY CPCMDs QUERY SUBCMD ACC
```

The resulting output would show that the QUERY ACCOUNT shares the same abbreviation. Thus, 3 characters for an abbreviation of "ACCUMULATOR" would be an illegal choice.

As indicated earlier in this section, if a command has multiple versions (IBM Classes) then all versions of the command must specify the same entry point. The DEFINE COMMAND and MODIFY COMMAND commands will not allow you to violate this requirement. However, if you had previously changed the entry point name associated with a command and defined a new command version using the new entry point name, you would be unwittingly creating a situation where you could not use the RESET option on the MODIFY COMMAND command to restore the CMDBKs to their original state. For this reason, you want to define all versions of a command prior to changing their entry point name to a new name.

Our next step in constructing the DEFINE COMMAND command is to decide when a command may be issued and what IBM classes and user privilege classes should be assigned to the command.

If the version of the command that you are defining is valid before the user has logged on then you can choose BEFORE_LOGON or ANYTIME, depending on whether it is valid after the user ID has logged on. For example, the MESSAGE command has a version that is valid ANYTIME, but the DIAL command is valid only BEFORE_LOGON. If either of these two options are chosen then PRIVCLASSANY must be chosen for the privilege class and this will generate an IBMCLASS of 0.

If the version of the command is valid only after a user has logged on then AFTER_LOGON should be specified. This is also the default. If you do not specify PRIVCLASSANY for this version then you would specify the PRIVCLASSES operand and the IBMCLASS operand. The IBMCLASS operand will define the IBM class to be associated with this version and PRIVCLASSES would define the initial mapping of the IBM class to the user privilege class allowed to issue this version of the command.

The default state for defining a command is that it is disabled. You can override that state by specifying the ENABLE operand or enable it by using the ENABLE COMMAND command.

Defining Your Diagnose Code

In this section we discuss the DEFINE command operands which are unique to Diagnose code processing.

After considering the ESM processing operands, the next step is to consider the privilege classes authorized to invoke your Diagnose code. You can specify either PRIVCLASSANY or specify the specific privilege classes that can invoke the Diagnose code.

The next two operands provide directives to the Diagnose code router concerning additional checks that should be performed prior to invoking the Diagnose code handler.

The INVAR operand specifies that CP does not process the DIAGNOSE code if the virtual machine that issues the DIAGNOSE code is in host access-register mode. CP returns a Special-Operation Exception program interrupt.

The INVXC operand specifies that CP does not process the DIAGNOSE code if the virtual machine that issues the DIAGNOSE code is in XC architecture mode. CP returns a Specification Exception program interrupt. XC architecture mode includes Enterprise System Architecture/Extended Configuration (ESA/XC) mode and z/Architecture Extended Configuration (z/XC) mode.

If your Diagnose code handler uses register 15 to pass results and directives to the Diagnose code router, you must activate this feature using the CHECKR15 YES operand. Otherwise, the values in register 15 are ignored on return to the Diagnose code router. See the description of the DEFINE DIAGNOSE command in *z/VM: CP Commands and Utilities Reference* concerning the CHECKR15 operand.

The default state for defining a Diagnose code is that it is disabled. You can override that state by specifying the ENABLE operand or enable it by using the ENABLE DIAGNOSE command.

Defining an Alias Command

CP restricts an alias command from referencing another alias command. When defining a new alias command, you must name a base command as its referenced command. For example, the command MSG is actually an alias for the MESSAGE command. MESSAGE is called a base command by elimination: because it is not an alias command, it must be a base command.

To determine if a command you want to define is an alias or a base command, use the QUERY CPCMDS command. This shows that MESSAGE is a base command, because it is not shown to be an alias command.

```
CP Q CPCMDS MESSAGE
Command: MESSAGE
  Status:      Enabled
  IBM Class:   A          PrivClasses: A
  CMBK Address: 004CFD28  Entry Point: HCPXMGMS
Command: MESSAGE
  Status:      Enabled
  IBM Class:   B          PrivClasses: B
  CMBK Address: 004CFD98  Entry Point: HCPXMGMS
Command: MESSAGE
  Status:      Enabled
  IBM Class:   0          PrivClasses: <ANY>
  CMBK Address: 004CFE08  Entry Point: HCPXMGMS
Ready
```

This shows that MSG is an alias command for MESSAGE.

```
CP Q CPCMDS MSG
Alias: MSG
  Alias Status:      Enabled
  Alias CMBK Address: 004D0118
  Command: MESSAGE
  IBM Class:        A          PrivClasses: A
  CMBK Address: 004CFD28  Entry Point: HCPXMGMS
Alias: MSG
  Alias Status:      Enabled
  Alias CMBK Address: 004D0188
  Command: MESSAGE
  IBM Class:        B          PrivClasses: B
  CMBK Address: 004CFD98  Entry Point: HCPXMGMS
Alias: MSG
  Alias Status:      Enabled
  Alias CMBK Address: 004D01F8
  Command: MESSAGE
  IBM Class:        0          PrivClasses: <ANY>
```

```
CMDBK Address: 004CFE08   Entry Point: HCPXMGMS  
Ready;
```

Once defined, an alias name cannot be reused. An alias command can be disabled, but it cannot be modified or eliminated until a SHUTDOWN or RESTART IPL is done.

Modifying a Command or Diagnose Code

The ability to alter attributes like privilege classes or the name of the routine to call is indispensable if you must tailor a z/VM system while avoiding modifications to IBM source files. You can redefine the following attributes of a CP command or a Diagnose code:

- Privilege class, or PRIVCLASSANY
- Name of the routine used to process the command or Diagnose code

To redefine CP commands and Diagnose codes during system initialization, use the MODIFY COMMAND and MODIFY DIAGNOSE statements in the system configuration file. For information about these statements, see [z/VM: CP Planning and Administration](#).

To dynamically redefine CP commands and Diagnose codes after system initialization, use the MODIFY COMMAND and MODIFY DIAGNOSE commands. For information about these commands, see [z/VM: CP Commands and Utilities Reference](#).

For more information about modifying commands and Diagnose codes, see the topic in [z/VM: CP Planning and Administration](#).

Disabling a Command or Diagnose Code

If for some reason, you need to disable a command or Diagnose code, you can do so using the DISABLE COMMAND and DISABLE DIAGNOSE commands. Especially for Diagnose codes, care should be taken to verify that disabling the Diagnose code does not break any other function. For example, disabling Diagnose code X'008' would bring most CMS users to a stand still because this is the Diagnose code used by CMS to pass CP commands for processing. Also, if you disable a base command, you are also disabling all aliases of that command.

Chapter 7. Defining and Using CP Message Repositories

This section describes:

- Why you would use message repositories
- How to create a message repository file
- How to load and unload message repository files
- How to produce messages using a repository

Why Use Message Repositories?

When you write CP code and you want to display error messages, you can put message text directly into the code. However, if you have many messages, your code can become cluttered and the size of the module will grow unnecessarily. Instead of coding message text directly in your code, you can store all your message text in a file called a repository. Then, to display a message, CP will retrieve the message text from this local repository on your behalf.

Having all message text in one central file has the following advantages:

- Message text does not clutter your code.
- You can access the same message from many modules without having to specify the message text each time.
- You can have your messages translated into other languages. You can then have your messages in any number of languages without changing your executable module.

For CP system messages, a source repository file is already built for you. It has a file ID of HCPMES REPOS. You can edit this file to view messages. You can also print a copy of the CP message file so you can refer to it when you want to call an existing CP message from your code.

Note: The file name of the CP message repository is different for languages other than English that are available on your system. For more information, see [z/VM: Installation Guide](#).

Components of a Message

A message can be logically divided into two components, the header (also called the code) and the text. The user can control which portion of a message they will receive using the message settings (e.g. SET EMSG TEXT, SET EMSG CODE, etc.). In addition, the two parts of a message can be further divided into subparts.

The message header, `xxxxxxnnns`, consists of three subparts:

xxxxxx

is the first six characters of the module ID that caused the message to be generated.

nnnn

is the message number. If the number is less than 1000 then only the last three digits are displayed, otherwise all four digits are displayed.

s

is the severity. This character is obtained from the message repository for that specific message.

The message text consists of the constant text portion and any substitution data that is passed when the message is generated. The constant text portion and location of the substitution data is defined for the message in the message repository.

Creating a Message Repository File

The message repository consists of a source file that is processed into a machine readable file (object repository file) that can be used by CP. You can create the source file using XEDIT.

When creating the file, we suggest that you specify support identification codes (SID codes) in columns 64 through 71 to identify who changed a line of the file and why. The SET SIDCODE subcommand in XEDIT will insert the SID code for you as you change the file. If you choose to use SID codes then you will specify the MARGIN option on the GENMSG command when you proceed to create the machine readable file.

The source file consists of:

- Comment records.
- A control line to indicate the control character used to identify substitution indicators.
- Message records which contain the message text along with any substitution indicators. The message records can identify multiple lines of message output.

For information on the format of the source file, see [Appendix H, “Understanding the CP Message Repository,”](#) on page 209.

Example of a Message Repository

The following example shows an external repository file, SPGMES REPOS.

```
SPGMES  REPOS  A1  F 80  Trunc=80 Size=14 Line=1 Col=1 Alt=0

00000 * * * Top of File * * *
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7...
00001 *
00002 * This is an example of a message repository file.
00003 *
00004 * This file was created using XEDIT.
00005 * You can code a file similar to this for your own installation.
00006 *
00007 $
00008 00050101E Invalid syntax; please issue command again
00009 001501 A Enter the number of copies you want:
00010 00250101I Function has completed
00011 00250201I Subroutine has completed
00012 01000101A Your program halted at label ABCD
00013 01000102A To quit the program, enter 'Q', or
00014 01000103A to continue, press the ENTER key
00015 * * * End of File * * *
```

Figure 4. Sample Repository - SPGMES REPOS

Here is a description of what this message repository contains:

Line

Content

1-6

Comment lines. Any line with an asterisk (*) in column 1 is a comment line.

7

The control line. The first nonblank character on this line (\$) defines the substitution character for messages. (For more information, see [“Producing Messages With Substitution”](#) on page 56.)

8

The first message in the repository is number 0005. It has only one version (columns 5 and 6) and is a 1-line message (columns 7 and 8). This message is issued in response to a user error, so the severity (column 9) is "E".

9

The second message in the repository is number 0015. Again, it has only one version (columns 5 and 6) and is a 1-line message (columns 7 and 8). We did not need to specify the message line number

because it has only one line. The message is requesting immediate action by the user so the severity (column 9) is "A".

10-11

The third message in the repository is number 0025. This message has two formats: depending on the error, either Function has completed (format 01) or Subroutine has completed (format 02) is displayed. These messages give the user information, so the severity (column 9) is "I".

12 - 14

The fourth message is number 0100. This message has only one format, but it spreads across three lines of the repository. Columns (7-8) show the line numbers of this message. The message is requesting immediate action by the user so the severity (column 9) is "A".

Generating the Object Repository File

After you create a message repository, you should check for incorrect entries in the message repository file, correct these errors, and then compile the message repository file into an object repository file. Use the GENMSG command to do the checking and compiling.

You can use the GENMSG command with the NOOBJECT option to check for errors in the message repository file. When you specify the NOOBJECT option, CMS only checks for errors. The message repository file is not compiled. When the message repository does not contain any errors, use the GENMSG command without the NOOBJECT option to compile the message repository file.

The GENMSG command with the NOOBJECT option does not create a TEXT file, it only creates a LISTING file. The LISTING file contains the messages returned from the GENMSG command. The GENMSG command without the NOOBJECT option creates two files. One file has a file type of LISTING. The other file has a file type of TEXT. The TEXT file contains the internal version of your message repository. The file name of the LISTING file and TEXT file is the same as the message repository source file name.

When you look at the LISTING file for information about an error in the message repository file, search for DMSMGC. The line containing DMSMGC describes the error.

See [z/VM: CMS Commands and Utilities Reference](#) for details on the GENMSG command.

Example 1

Enter the following GENMSG command to check for errors in your message repository file called SPGMES REPOS:

```
GENMSG SPGMES REPOS A SPG (NOOBJECT MARGIN 63
```

In this example, SPGMES REPOS A is the file ID of the message repository you created. The operand SPG is the *applid*, which is the operand used to identify your application. This application identifier must be three characters long. Be sure to record the application identifier you choose. You will need to reference it as the component ID when you specify the HPCONSL macro to generate the message from your code. The MARGIN 63 operand indicates that the data for the message repository is contained in columns 1 through 63. In this example, we assume that we specified the SID codes in columns 64 through 71.

Once the errors are corrected in the message repository file, enter the following GENMSG command to compile SPGMES REPOS:

```
GENMSG SPGMES REPOS A SPG ( CP MARGIN 63
```

The CP operand indicates that the object file should be in a format useable by CP. CP's message repository format is different from the format used by CMS and other components.

Example 2

Enter the following GENMSG command to check for errors in your message repository file called SPGMES REPOS and to compile SPGMES REPOS:

```
GENMSG SPGMES REPOS A SPG (CP
```

In this example, the MARGIN 63 option is not used. The data for the message repository is expected to be in columns 1 through 71. Once again, the CP operand is specified so that the correct format of the object directory is created.

If there are incorrect syntax statements in the message repository file, correct the errors and issue the command again.

Loading and Unloading Message Files

Once you have created the object repository, you will need to make it accessible by CP code. This will consist of moving the file to CP accessed minidisks, loading it into the system execution space, and associating the message repository with a component ID and language.

As indicated in [Chapter 4, “Loading Dynamically into the System Execution Space,”](#) on page 33, the message must be loaded into the system execution space prior to attempting to use it in CP code. In order to access the file as part of the load function, CPXLOAD requires that the object repository file reside on a CP-accessible minidisk.

Once the object repository file is on the CP-accessed minidisk, you can invoke CPXLOAD to load it into the system execution space. Continuing with our example repository of SPGMES, we would issue the following command:

```
CPXLOAD SPGMES TEXT * TEMPORARY NOCONTROL
```

SPGMES TEXT is the file to be loaded and it is located on one of the CP accessed disks. The file is loaded TEMPORARY so that it can be replaced if necessary. No control entry point is associated with this load.

Once loaded into the system execution space, the next task is to associate the message with a component ID so that invocations of HCPCONSL for that component will find the repository. Unlike exits, we must load the repository using CPXLOAD prior to associating the entry point.

```
ASSOCIATE MSGS COMP SPG EPNAME SPGMES
```

SPGMES is the entry point that is to be associated with component SPG. The language is not specified on the ASSOCIATE command because that information was specified when the object repository file was created by GENMSG.

If you wish to replace an existing message repository then you will need to disassociate the repository file and unload it prior to replacing it. An alternative is to associate a different entry point with the component for the language being replaced (see example 2 below).

Example 1

```
DISASSOCIATE MSGS COMP SPG LANGUAGE UCENG
```

As with the ASSOCIATE MSGS command, we specified the component ID but instead of an entry point we specified the language which we will disassociate. This is necessary because the same entry point could have been associated with different languages. We must disassociate the object repository with each language and component that it is associated prior to unloading it.

Example 2

```
ASSOCIATE MSGS COMP SPG REPLACE EPNAME SPGMET
```

The REPLACE operand on the ASSOCIATE MSGS command indicates that the existing repository file for the language contained in SPGMET is to be replaced.

CPXUNLOAD will unload the message repository from the system execution space. For further information on unloading files, see [Chapter 8, “Unloading Dynamically from the System Execution Space,”](#) on page 61.

Producing Messages from a Repository

Now you know how to add a message repository to the system. The next step is to discuss how the system decides which repository to use and how you can code invocations of a CP macro to generate the messages.

HCPCONSL is the macro that encapsulates the calls to message routines that generate messages or read input. You should consult the macro prologue for further information about its use. The examples in this section will illustrate the most common usage of the macro. See [Chapter 3, “Creating a Dynamically Loaded Routine,”](#) on page 11 for a discussion of the HCPCONSL macro.

How Does CP Choose Which Repository to Use?

When searching the message repositories for a message, CP limits its search by asking two questions.

Question 1:

Which component ID is being used on the invocation of HCPCONSL? CP will search only the repositories associated with the component ID specified on the HCPCONSL invocation. If the COMPID= operand is not specified then the component ID defaults to the system repository, COMPID=HCP.

Question 2:

Is there a repository for the current language being used by the receiver of the message? Only the correct language message repository will be searched.

When the appropriate repositories are located (there can be more than one entry point associated with a specific message repository) then the repositories are searched in the order in which the repositories were associated with the component by the ASSOCIATE MSGS command.

For component HCP, the local repositories are searched prior to searching the default system repository. This allows you to override CP message texts with your own message texts.

If the message is not found in the message repositories then the message will be displayed without the message text but may include any substitution data. The severity field in the message header is set to 'E' because the severity code could not be obtained from the message repository. Of course, the resulting message output depends upon both the HCPCONSL operands and the user message settings. [Table 4 on page 55](#) shows the output when a message is found and the output when a message is not found in the repository.

Table 4. Results if a message is found or not found in a repository.

EMSG Setting	Substitution data	Message Found	Message Not Found
CODE	n/a	message header only	message header only
ON	no	message header and message text	message header only
ON	yes	message header and message text with substitution data	message header with substitution data separated by blanks
TEXT	no	message text only	blank line
TEXT	yes	message text with substitution data	substitution data separated by blanks

Producing Messages Without Substitution

The following is an example of an HCPCONSL invocation for the repository in [Figure 4 on page 52](#) to display a message without substitution.

```

      HCPCONSL (WRITE),                                X
              COMPID='SPG',                            X
              REPOSNUM='MS000501'
      B      EXIT                                     Skip around constants
      SPACE 1
MS000501 EQU X'00000501'      Message number/version in hex

```

Figure 5. Sample Assembler Code Accessing SPGMES REPOS from module HCPXXX

The HCPCONSL invocation:

WRITE

indicates that a console write is requested.

COMPID=

indicates the component ID whose repositories are to be searched for the specified message number and version.

REPOSNUM=

indicates the equate containing the message number and version.

The result of the previous HCPCONSL invocation is:

```
HCPXXX0005E Invalid syntax; please issue command again
```

Producing Messages With Substitution

In the SPGMES REPOS example, the text for each message is the same every time the message is displayed. However, you will probably want to have some message texts that are similar, but say different things depending on the situation. For example, you might have a message that says:

```
Invalid option: GO
```

But you also want to have these messages in your repository:

```
Invalid option: FILE
Invalid option: RUN
Invalid option: STOP
```

You do not need four separate messages in your repository. Instead, you can create a single message text that contains a substitution indicator. CP will substitute different information using this substitution indicator. For example, your repository could look like this:

```

JCRMES    REPOS    A1  F 80  Trunc=80 Size=6 Line=1 Col=1 Alt=0

===== * * * Top of File * * *
|...+...1...+...2...+...3...+...4...+...5...+...6...+...7...
===== *
===== * This is an example of a message repository file that
===== * uses simple substitution.
===== *
===== $
===== 020001 E Invalid option: $1
===== * * * End of File * * *

```

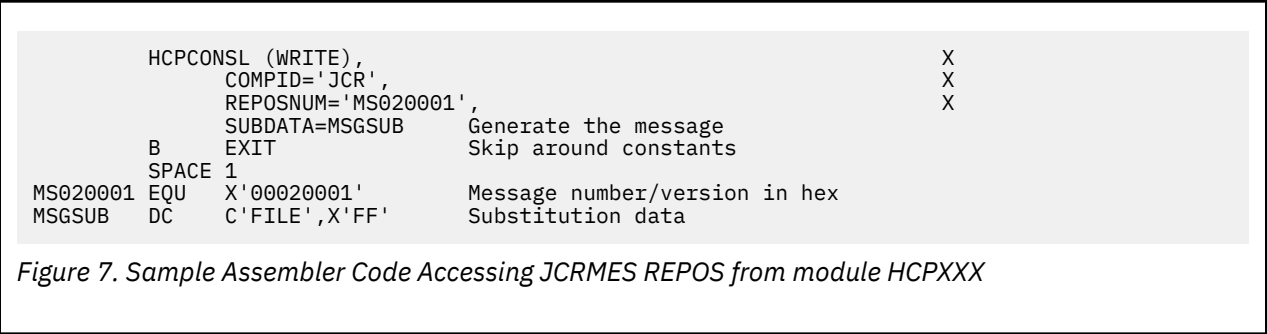
Figure 6. Sample Repository - JCRMES REPOS

Messages that require substitutions have parameters in a form defined by the user (for example, \$1, \$2 ...). These parameters show the placement of the substitutions and their order. The first character in the first noncommentary record of the source repository defines the substitution character. This character cannot be a DBCS character.

Here are some rules about substitutions:

- A substitution can be a single word, a phrase, or an entire sentence.
- A substitution can go anywhere within a message.
- You can have more than one substitution per message.
- Trailing blanks from the substitution data are removed when non-column format is used.

The following is an example of an HCPCONSL invocation for the repository in [Figure 6 on page 56](#).



The HCPCONSL invocation:

- WRITE**
indicates that a console write is requested.
- COMPID=**
indicates the component ID whose repositories are to be searched for the specified message number and version.
- REPOSNUM=**
indicates the equate containing the message number and version.
- SUBDATA=**
indicates the variable containing the message substitution data. The end of this data is signaled by the x'ff' value.

The result of the previous HCPCONSL invocation is:

```
HCPXXX200E Invalid option: FILE
```

Producing Messages With Column Format

Substitution can also be used to build messages with column alignment. The repository in [Figure 8 on page 58](#) illustrates the use of substitution in this manner.

```
JAFMES   REPOS   A1  F 80  Trunc=80 Size=14 Line=1 Col=1 Alt=0

00000 * * * Top of File * * *
      |...+....1...+....2...+....3...+....4...+....5...+....6...+....7...
00001 *
00002 * This is an example of a message repository file made via XEDIT
00003 * and maintain alignment of data.
00004 * You can code a file similar to this for your application.
00005 *
00006 $
00007 74000101R      PREV OWNER  CURR OWNER  NEXT OWNER
00008 74000102      -----
00009 74000103  USERID $1          $3          $5
00010 74000104  NODEID $2         $4          $6
00011 * * * End of File * * *
```

Figure 8. Sample Repository - JAFMES REPOS

Here is a line-by-line description of what this repository contains:

Line number(s)
Explanation

1 - 5

Comment lines.

6

The control line.

A dollar sign (\$) is the substitution character.

7 - 10

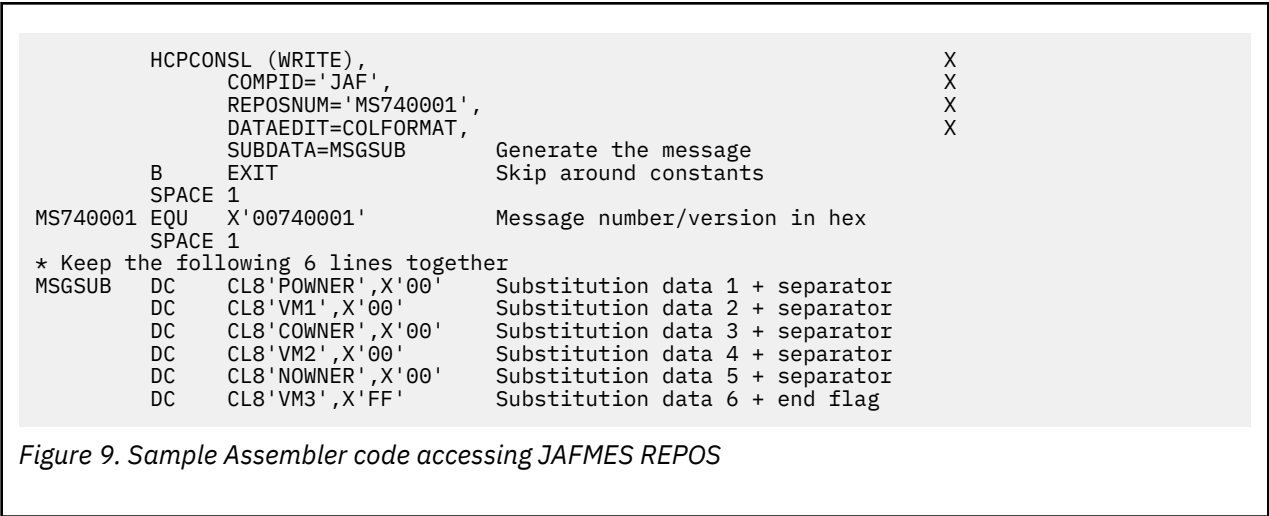
Message number is 7400. This message has only one format and will be displayed as four lines. There are six substitution indicators which should line up as specified. We made certain that the substitution data will fit within the space set aside for the data. This was done by specifying enough blank space so that the substitution data may replace the indicator and the extra blank space.

Because the message number is in the range of 7000-7999, CP will display the message without the message header. The 7000 series messages are normally used for responses that do not utilize a header.

Here are some rules about column substitutions:

- The message in the repository must have a number of blanks following the substitution data symbol, '\$nnn', such that the length of the symbol plus the number of blanks is equal to or greater than the maximum length of the substitution data.
- If the length of the substitution data is less than the length of the substitution symbol, '\$nnn', blanks will be placed after the substitution data to make it the same length as the '\$nnn'.
- If a message substitution place holder is used, blanks will be filled in that substitution field.

The following is an example of HCPCONSL invocations for the repository in [Figure 8 on page 58](#).



The HCPCONSL invocation:

- WRITE**
indicates that a console write is requested.
- COMPID=**
indicates the component ID whose repositories are to be searched for the specified message number and version.
- DATAEDIT=**
indicates the format of the data which in this case is column formatted.
- REPOSNUM=**
indicates the equate containing the message number and version.
- SUBDATA=**
indicates the variable containing the message substitution data. The end of this data is signaled by the x'ff' value.

The output from this HCPCONSL invocation will look like this:

	PREV OWNER	CURR OWNER	NEXT OWNER
	-----	-----	-----
USERID	POWNER	COWNER	NOWNER
NODEID	VM1	VM2	VM3

When You Cannot Use a Message Repository

- There are situations when you cannot use a message repository or HCPCONSL. These situations include:
- Points in the code where you cannot tolerate a loss of control.
 - From modules that do not have a savearea.
 - From modules that run under CMS (e.g. DIRECTXA) or are standalone routines (e.g. HCPSAL).
 - Points early in initialization processing or near the end of system termination processing.
 - Modules where R11 does not point to a VMDBK.

Chapter 8. Unloading Dynamically from the System Execution Space

This section describes how to remove your dynamically loaded routines from the system execution space.

What May Be Unloaded

Any module that was loaded by CPXLOAD TEMPORARY may be unloaded by CPXUNLOAD. Usually, any such module may be unloaded at any time. This would include modules that contain external symbols used as entry points for commands, Diagnose codes, or CP exit routines. If the CPXLOAD was performed with the PERMANENT operand, the CPXUNLOAD command will be rejected with this error message:

```
HCP2769E CPXUNLOAD for load ID $1 has been rejected;
         the requested load ID was loaded as PERMANENT
```

There is nothing to be done, except to remember to perform a CPXLOAD TEMPORARY the next time that you load the module. This will need to be done after the next IPL.

Even if loaded by CPXLOAD TEMPORARY, there still are reasons why the CPXUNLOAD command could fail. An error message from CPXUNLOAD may indicate that the CPXUNLOAD command may not be issued at this time:

```
HCP2769E CPXUNLOAD for load ID $1 has been rejected;
         the requested load ID is still in use by a
         system service
```

This indicates that something that was loaded as part of the indicated load ID cannot be unloaded at this time without exposing CP to catastrophe. The system service that may be involved could be message processing.

Some external symbol from the load ID has been associated with message processing by ASSOCIATE MESSAGE. The external symbol must be removed from this association before CPXUNLOAD can be permitted.

To see if the external symbol is in use for message processing, issue the following command:

```
CP QUERY CPLANGLIST ASSOCIATED
```

Removal from use for message processing can be accomplished by either of the following commands:

- ASSOCIATE MESSAGE with the REPLACE operand, without including the external symbol among the EPNAME list of external symbols
- DISASSOCIATE MESSAGE

Another exception would be when the CPXUNLOAD command is rejected by the CONTROL routine. If this happens, this message should be displayed:

```
HCP2769E CPXUNLOAD for load ID $1 has been rejected;
         the request was denied by the control entry
         point
```

This indicates that the control routine specified on the CPXLOAD request for this load ID has decided that the CPXUNLOAD request must not be processed. A well behaved control routine will indicate why it rejected the CPXUNLOAD request. Because this decision is completely under the control of the control routine, CP has no idea why the request was rejected. The only thing that CP will do is honor the control routine's decision. You will need to examine any messages from the control routine to determine your next action.

CPXLOAD Load ID Number

The syntax for the CPXUNLOAD command requires the load ID assigned by CPXLOAD. This number can be remembered from the time of the CPXLOAD request, or can be determined from QUERY CPXLOAD responses. However it may be determined, supply it on the CPXUNLOAD command.

When to Use ASYNCHRONOUS and SYNCHRONOUS

A CP command is normally processed under the control of the user's Virtual Machine Definition Block (VMDBK). While processing a command, no other activity for that VMDBK is allowed. This means that any other command that the user may issue is held waiting until all prior commands issued by the user have completed.

In a busy system, a CPXUNLOAD command may take longer than the user is willing to wait. To complete a CPXUNLOAD request, CP must prevent all new attempts to use the module, and then wait until all present uses complete. Then, finally, CP can remove the module from storage.

To avoid such a delay, the user may use the CPXUNLOAD ASYNCHRONOUS operand (which is the default). This operand tells CP to process the command on the system's VMDBK, not on the user's VMDBK. By switching the command to the system's VMDBK, CP can make the user's VMDBK immediately available for more commands. The user at the terminal is not delayed from doing more work.

If the CPXUNLOAD ASYNCHRONOUS operand is used by an EXEC or by a program, this also means that the EXEC or program continues executing beyond the CPXUNLOAD command. If the CPXUNLOAD processing generates an informational message or an error message, there is no program ready to receive it. Therefore, all messages will be sent to the user's terminal. This also means that the EXEC or program will not know if the CPXUNLOAD command succeeded or failed.

The best use of the SYNCHRONOUS and ASYNCHRONOUS operands is this:

- If the user is issuing the command at the terminal, use (or allow to default to) the ASYNCHRONOUS operand. The user may continue working while the CPXUNLOAD command continues.
- If the user is issuing the command by an EXEC or a program, use the SYNCHRONOUS operand. The EXEC or program can then look for messages or error return code from the command and make decisions about how to proceed.

Chapter 9. Understanding the CP Parser

The CP parser simplifies the process of developing new CP commands and configuration file statements. Rather than writing code to implement the logic for parsing an input string, you can make use of the CP parser facilities to handle that effort for you.

By using a set of syntax definition macros, you define your syntax and generate the data structures that represent that syntax. The CP parser can then use that information to parse an input string (for example, a command). Based on how you define your syntax, the parser can perform tasks such as:

- Generating messages for certain error situations
- Converting token values for you
- Saving information in data areas of your choice
- Driving a post-processor routine you provide to handle additional processing

Introduction to the Syntax Definition Macros

The parser facility provides the following set of syntax definition macros that you can use to define the rules of your syntax:

- HCPCFDEF

This macro is used to mark the start of a given command or configuration file statement. You use this macro to define the command or statement verb. The minimum abbreviation can be specified either through the transition from upper to lower case in the verb (e.g. 'DISconnect' has a minimum abbreviation of 3 characters) or by means of a keyword parameter. You can also use this macro to indicate the name of a post-processor routine you want called if the statement or command entered passed all syntax checks.

For a description of all of the parameters available on this macro refer to [“HCPCFDEF: Command/Config File Statement Definition Macro” on page 190.](#)

- HCPSTDEF

Each HCPCFDEF macro has one or more HCPSTDEF macros associated with it. The HCPSTDEF macro identifies the additional states or pathways that exist in your command or statement syntax. There are keyword parameters to describe properties associated with the tokens that are parsed. For example, is the token optional; is at least one token match out of a list of possible choices required; is it acceptable to find no tokens on the input line. There are keyword parameters that tell the parser what HCPSTDEF macro to look at next when a token match occurs. There are also keyword parameters that can be used to identify error message equates to be used for the certain missing token situations.

For a description of all of the parameters available on this macro refer to [“HCPSTDEF: Parser State Definition Macro” on page 193.](#)

- HCPTKDEF

Each HCPSTDEF macro has one or more HCPTKDEF macros associated with it. The HCPTKDEF macro defines what a token can be and what the parser should do when that token is found. There are keyword parameters that let you tell the parser the type of validation you want performed on the token. For example, is the token a valid hex number; is the token within a specified range of numeric values; is the token something that would be a valid file mode. There are a variety of keywords to tell the parser how to store information into plist areas passed to the parser. You can indicate error message equates to use for certain error conditions. There are keyword parameters that can be used to tell the parser what HCPSTDEF macro to look at next when a token match occurs.

For a description of all of the parameters available on this macro refer to [“HCPTKDEF: Parser Token Definition Macro” on page 196.](#)

- HCPDOSYN

The HPCPDEF, HCPSTDEF and HCPTKDEF provide a way to set global variables, but they do not generate any actual data structures. The HCPDOSYN macro takes the global variables and creates the data structures that represent your syntax definition. The HCPDOSYN macro identifies the plists that are used for storing information. You can provide error message equates to be used for a variety of syntax errors that may be detected. The HCPDOSYN macro also provides a way for you to tie together syntax definitions that have been split over several modules.

For a description of all of the parameters available on this macro refer to [“HCPDOSYN: Parser Syntax Table Generator Macro”](#) on page 191.

Understanding How to Code for the Parser Using an Example

In this example we will cover the various steps necessary to develop a syntax table for use with the CP parser and the related processing. Ours is a 10 Step Process:

Steps

What to do

- 1** Develop the command syntax
- 2-5** Determine the locations of the HPCPDEF, HCPSTDEF and HCPTKDEF macros
- 6-7** Further develop the HCPSTDEF and HCPTKDEF macros to transition flows and error processing.
- 8** Develop the HCPDOSYN macro.
- 9** Write the code to call the parser.
- 10** Write the commands to activate the code.

Step 1: Develop the Syntax Railroad Track Diagram

We will not go into great detail on railroad track diagrams. Instead, we will use the railroad track diagram as a tool to develop our command syntax. See [“Syntax, Message, and Response Conventions”](#) on page xiii for a discussion of railroad track diagrams.

Let's begin with a very simple command. From there we will construct more complex versions of our command until we get to one that is sufficiently complex to illustrate various aspects of the parser. You should note that we could have easily implemented the simplest command and added the additional complexity later. One of the great strengths of the CP parser is the ease with which it can handle new syntax.

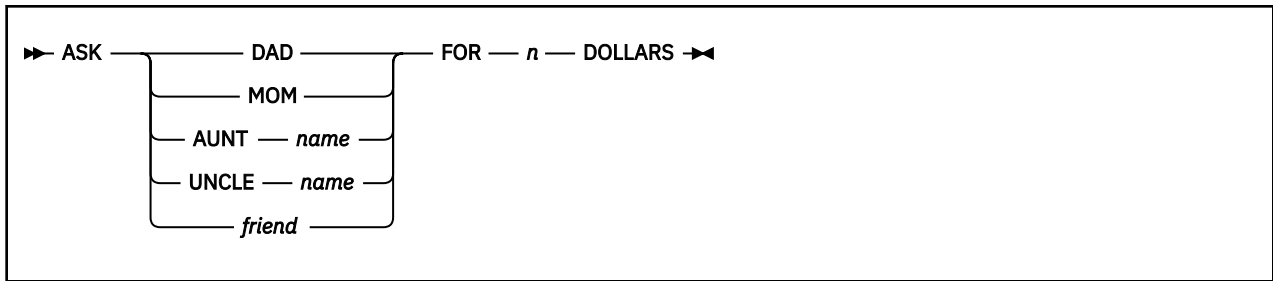
Our simplest railroad track syntax will be a command to ask Dad for n dollars. Of course, we can show you how to code the command syntax to request the dollars but we cannot show you how to get the dollars from Dad.

►► ASK — DAD — FOR — n — DOLLARS ◄◄

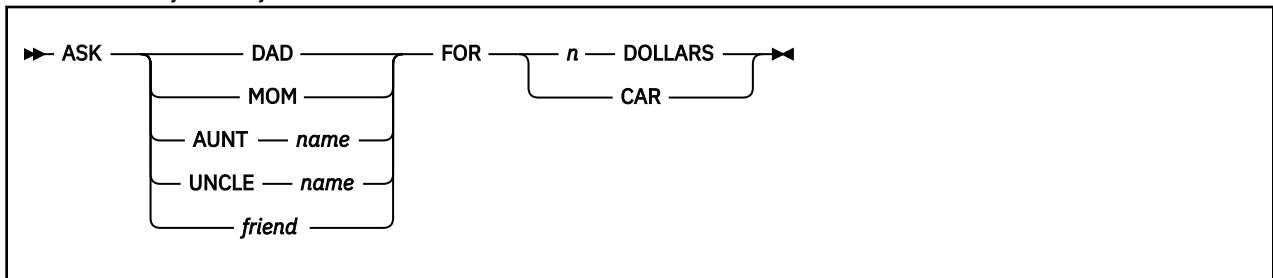
We expand this to give a choice of whom to ask for the money.

► ASK — DAD — MOM — FOR — n — DOLLARS ►

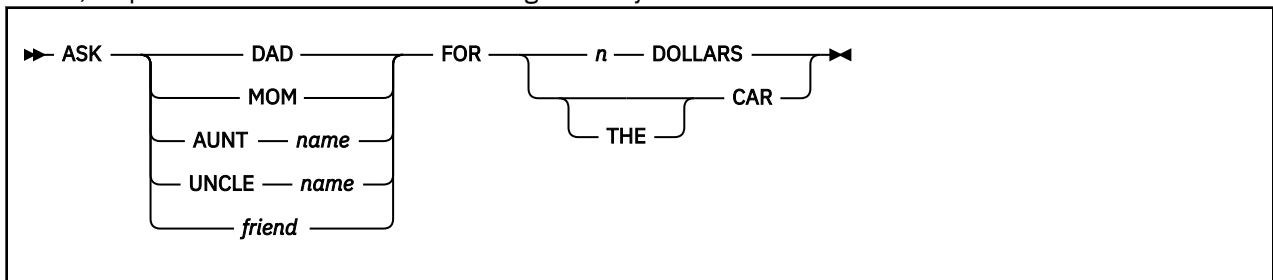
Let's expand this even further to ask additional people.



Now that we have grown who to ask, let us expand our horizons and think about a way to ask for a car instead of only money.



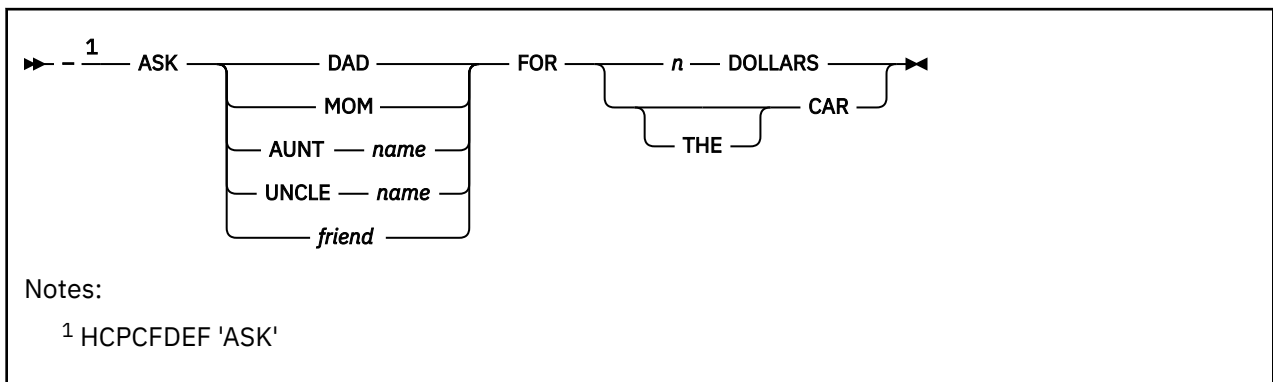
Finally, let's improve the command so that it is closer to an actual language. We will do this by adding in the ability to specify the word "THE" if desired. In this example, "THE" is affectionately known as a "noise word"; its presence or absence adds nothing to the syntax.



This, then, is the syntax that we will describe in the parser macros.

Step 2: Assign HCPCFDEF Macro to the First Token

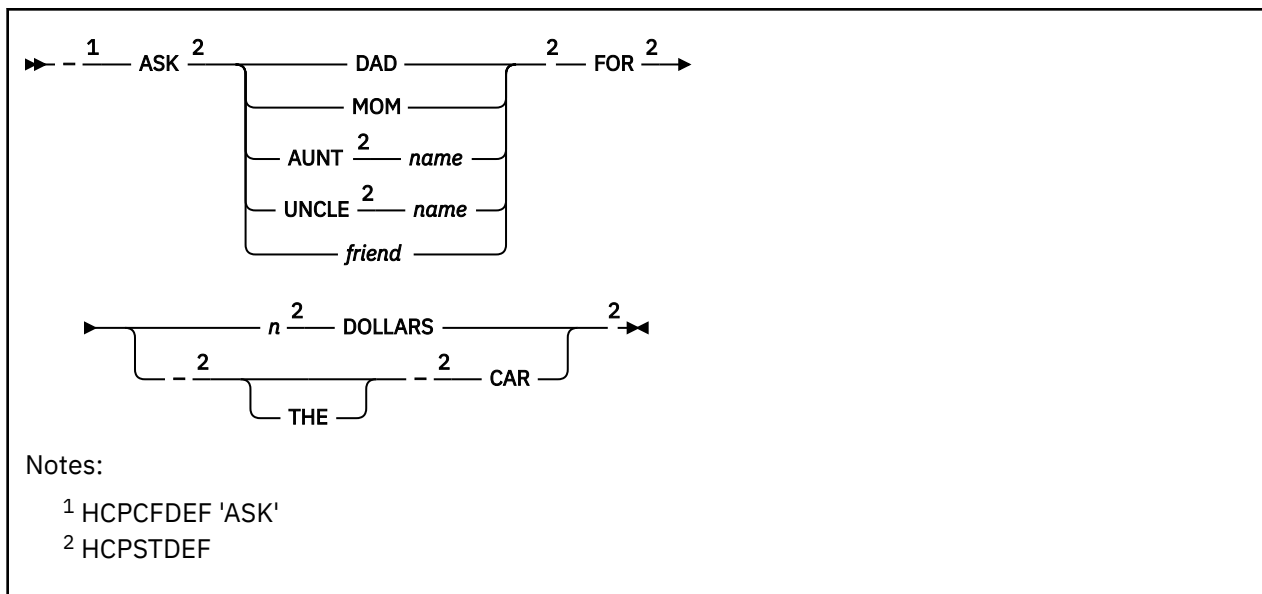
The HCPCFDEF macro starts the syntax macro definition. We list the command name with this token. So that you can follow our example, we will place "CF" at the location in the railroad track diagram to show that we have coded the HCPCFDEF macro.



Step 3: Mark Alternative Path Locations

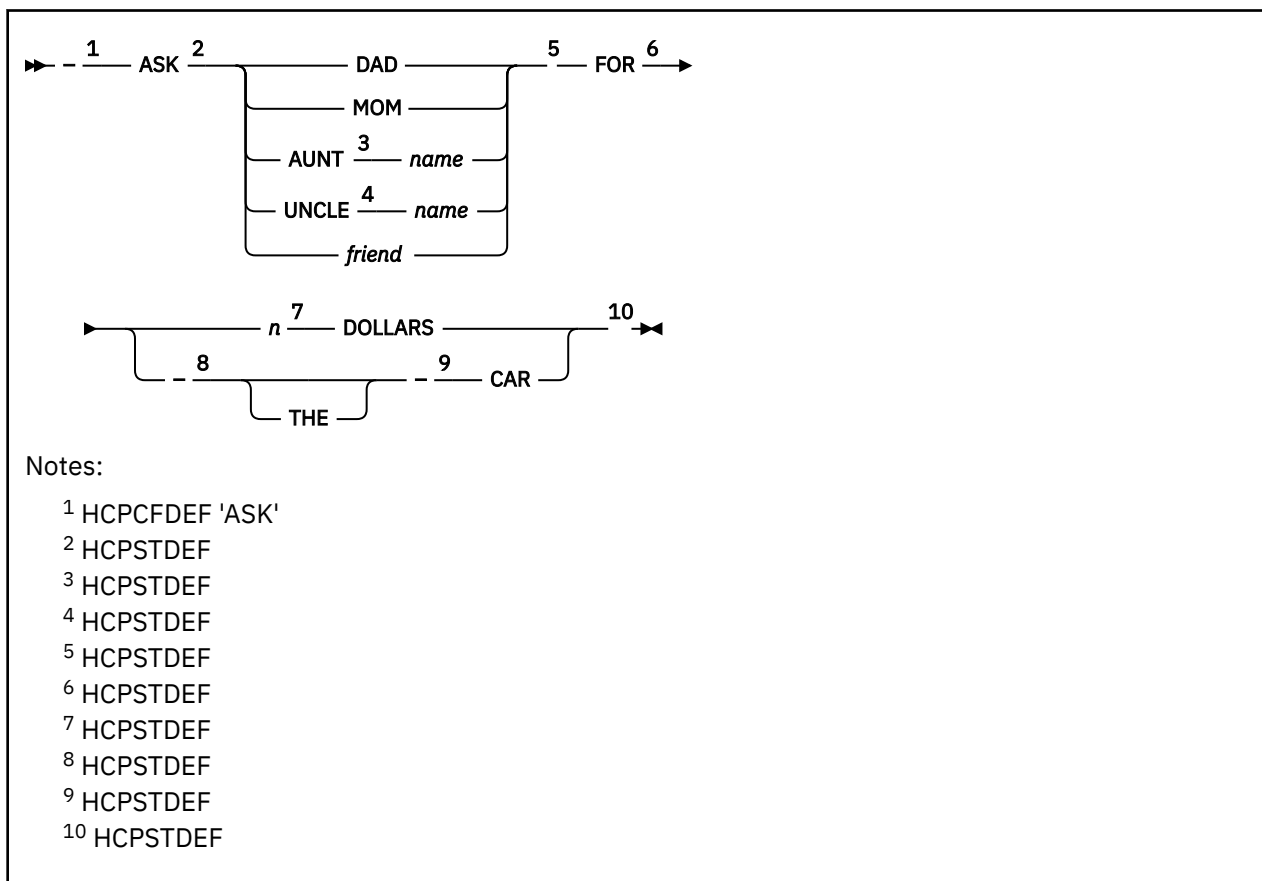
Now, we want to determine the various points on the railroad track diagram where we must handle alternate paths. Remember that at every word is an implicit path that can be thought of as "this is wrong". So, even when it appears that there is no possible alternate path (as in the path 'AUNT-name'), there is

always the implicit path to the end of parsing and a possible error message. Mark such places. Also, we want to mark the end of the railroad track syntax.



Step 4: Assign HCPSTDEF Macro to Alternative Path Locations

Assign HCPSTDEF macro to each location that was marked as an alternate path.



Step 5: Assign HCPTKDEF Macro to Each Token

For each HCPSTDEF (state definition) we want to code HCPTKDEF macros that indicate the possible tokens that can be encountered and the action to perform. Remember that every HCPSTDEF requires at least 1 HCPTKDEF, even if only to specify HCPTKDEF TYPE=NULL to indicate there is no token.

Five HCPTKDEF macros are added to the ST_1 HCPSTDEF macro. They describe the possible choices (MOM, DAD, ...).

As with REXX SELECT statements or CASE statements in other languages, the first statement that satisfies the "when" portion will be processed. For example, if we had coded the HCPTKDEF for *friend* (TYPE=TOKEN) as the first HCPTKDEF, then none of the others would be processed because TYPE=TOKEN says that a match occurs if any token is found. So, we would match TYPE=TOKEN for 'DAD', 'MOM', 'AUNT' and 'UNCLE'. Because we do not want this to occur, we are careful in placing the HCPTKDEFs to avoid such confusion.

```
ST_1      HCPSTDEF
          HCPTKDEF 'DAD'
          HCPTKDEF 'MOM'
          HCPTKDEF 'AUNT'
          HCPTKDEF 'UNCLE'
          HCPTKDEF TYPE=TOKEN           For 'friend'
```

Let us code the rest of the HCPTKDEF macros for the other HCPSTDEF macros ST_2 through ST_8. You should notice that in HCPTKDEF for ST_5 we are using another conversion type, DEC. This tells the parser that the token must be a decimal character value for a match to occur and that the token should be converted from decimal characters to their binary value when the value is saved.

```
ST_2      HCPSTDEF
          HCPTKDEF TYPE=TOKEN           For 'name'
ST_3      HCPSTDEF
          HCPTKDEF TYPE=TOKEN           For 'name'
ST_4      HCPSTDEF
          HCPTKDEF 'FOR'
ST_5      HCPSTDEF
          HCPTKDEF TYPE=DEC              For 'n'
ST_6      HCPSTDEF
          HCPTKDEF 'DOLLARS'
ST_7      HCPSTDEF
          HCPTKDEF 'THE'
ST_8      HCPSTDEF
          HCPTKDEF 'CAR'
```

We code the HCPTKDEF for the final HCPSTDEF as TYPE=NULL so that the parser knows that we should not expect any additional operands when processing this HCPSTDEF state.

```
ST_9      HCPSTDEF HCPTKDEF TYPE=NULL
```

Step 6: Add the Keywords for the Macros

Our next step is to provide the necessary information to the parser so that it knows how to flow control from the HCPCFDEF macro and among the HCPSTDEF macros.

Since we plan to process the command directly after the parser returns control to our program, we have no post-processor subroutine. Therefore, we code PROCESS=NONE on the HCPCFDEF macro. We will define default transition flows on the HCPSTDEF macros to indicate what happens if any HCPTKDEF has a match. We will also define specific transition flows on the HCPTKDEF macros to inform the parser where control flows if a match occurs with that HCPTKDEF macro. Both macros use the operand NEXT= to define the transition.

We must also indicate to the parser whether a match is required in order for a transition to occur to another HCPSTDEF macro. This is handled using the REQMATCH= operand on the HCPSTDEF macro.

We can also provide the parser with additional rules concerning conflicting options so that it can detect those conflicts for us. The CONFLICT= operand on the HCPTKDEF allows us to indicate an arbitrary numeric value (from 1 to 255) which indicates a possible conflict. When the parser detects that it has

a match on a HCPTKDEF macro it will compare the conflict values on that HCPTKDEF macro with any accumulated conflict values. If the value has already been saved, then we have a conflict and an error will be generated. Otherwise, there is no conflict. The parser adds any conflict values from this HCPTKDEF macro to the accumulated conflict values for later conflict checking, and then continues processing.

Let us update the ST_1 HCPSTDEF macro and related HCPTKDEF macros. You should note that we indicate on the HCPSTDEF macro that a match is required for one of the possible tokens (DAD, MOM, AUNT, UNCLE, any non-blank word). We add the NEXT= operands to indicate where parsing flows when any of the HCPTKDEF macros match. We also use the CONFLICT= operand to *friend* HCPTKDEF macro because we want the parser to look for and to reject what we have decided as an unacceptable combination (in this case, asking a friend for money).

```

HCPCFDEF 'ASK',PROCESS=NONE

ST_1      HCPSTDEF  REQMATCH=YES
          HCPTKDEF  'DAD',NEXT=ST_4
          HCPTKDEF  'MOM',NEXT=ST_4
          HCPTKDEF  'AUNT',NEXT=ST_2
          HCPTKDEF  'UNCLE',NEXT=ST_3
          HCPTKDEF  TYPE=TOKEN,NEXT=ST_4,    For 'friend'
          CONFLICT=1                        Do not ask for money

```

We update the HCPSTDEF/HCPTKDEF groups for ST_2 through ST_4 in the same manner. Notice that we did not specify a NEXT= operand for the ST_4 group. We can do this because the default transition is to the next HCPSTDEF/HCPTKDEF group that follows the current group. Thus, a match in ST_4 will flow into ST_5.

```

ST_2      HCPSTDEF  REQMATCH=YES
          HCPTKDEF  TYPE=TOKEN,NEXT=ST_4    For 'name'

ST_3      HCPSTDEF  REQMATCH=YES
          HCPTKDEF  TYPE=TOKEN,NEXT=ST_4    For 'name'

ST_4      HCPSTDEF  REQMATCH=YES
          HCPTKDEF  'FOR'

```

ST_5 shows an interesting series of checking. Should a match occur with the HCPTKDEF TYPE=DEC, then we will flow to ST_6. But a match is not required and we would flow to ST_7 if a match does not occur. This illustrates the use of the NEXT= operand on the HCPSTDEF macro in conjunction with the HCPTKDEF macro.

CONFLICT=1 is specified. Had the parser earlier matched the HCPTKDEF for 'friend', it would have saved the CONFLICT=1 value from that HCPTKDEF. If the parser now matches this HCPTKDEF TYPE=DEC macro, the parser would detect the conflict and generate the error message specified by the HCPDOSYN macro keyword CONFLICT=. Thus, we would delegate to the parser the work to enforce the rule that we do not ask friends for money.

In addition, we have added a new operand on the HCPTKDEF macro. The RANGE= operand informs the parser of additional rules that it needs to enforce. In our case, the token must be a decimal value, conflict 1 must not be on (do not ask a friend for money) and the value of token must be at least 1 and no more than 10,000.

```

ST_5      HCPSTDEF  REQMATCH=NO,NEXT=ST_7
          HCPTKDEF  TYPE=DEC,NEXT=ST_6,    For 'n'
          CONFLICT=1,                        Do not ask friend
          RANGE=(1,10000)                    No more than 10,000

```

ST_6 informs the parser that DOLLARS is a required token following the decimal value detected by ST_5. If there is no token from the input so that the parser has nothing to check, then error message 6704-01 should be generated from the message repository.

```

ST_6      HCPSTDEF  REQMATCH=YES,MISSING=MS670401
          HCPTKDEF  'DOLLARS',NEXT=ST_9

```

ST_7 and ST_8 use the operands that we have already discussed, so we have nothing wonderful to say about this segment of the syntax table. You may wish to note that they have been updated for the transition flow between states.

```
ST_7    HCPSTDEF REQMATCH=NO,NEXT=ST_8
        HCPTKDEF 'THE',NEXT=ST_8

ST_8    HCPSTDEF REQMATCH=YES
        HCPTKDEF 'CAR',NEXT=ST_9
```

The operand FINISH=YES marks the end of the syntax table. ST_9 informs the parser that, if we reach this HCPSTDEF, we are to end here and that no operands should be left to be processed.

```
ST_9    HCPSTDEF REQMATCH=NO,FINISH=YES
        HCPTKDEF TYPE=NULL
```

Step 7: Develop the Storage Area Definitions

While it is nice for the parser to validate the syntax of a command or statement, it is even nicer that it can update storage areas with what has been specified. HCPTKDEF supports the STORE=, ORFLAG= and ANDFLAG= operands to store data, 'OR' flag bits to turn them on, and 'AND' flag bits to turn them off, respectively. The target location for these storage operands would need to be described in an assembler language DSECT.

The DSECT might appear as follows. In this example, this DSECT is made the primary output plist by the PLBASE= keyword on the HCPDOSYN macro, which we will discuss in a later step. Here we see the definition of field names and equates to be used by the STORE=, ORFLAG= and ANDFLAG= operands.

```
ASKDSECT DSECT
HOWMUCH DS F
CWHO DS CL15
FROM DS B
FAUNT EQU X'80'
FDAD EQU X'40'
FFRIEND EQU X'20'
FMOM EQU X'10'
FUNCLE EQU X'01'
WHAT DS B
FCAR EQU X'20'
FUSD EQU X'02'
LENDSECT EQU *-ASKDSECT
```

Our expansion of the parser macros has grown with the addition of the new operands. ST_1, ST_6 and ST_8 show some simple OR operations using the ORFLAG= operand (for example, the bit defined by the FDAD equate would be 'OR'ed into the FROM field). ST_1, ST_2, ST_3 and ST_5 illustrate the use of the STORE= operand to indicate to the parser where it should store the token or converted data that it has encountered. The stored result is not allowed by the parser to be longer than the specified output field. An error message would be generated.

```
CFDEF    HPCFDEF 'ASK',PROCESS=NONE

ST_1     HCPSTDEF REQMATCH=YES
        HCPTKDEF 'DAD',NEXT=ST_4,
            ORFLAG=(FROM,FDAD)
        HCPTKDEF 'MOM',NEXT=ST_4,
            ORFLAG=(FROM,FMOM)
        HCPTKDEF 'AUNT',NEXT=ST_2,
            ORFLAG=(FROM,FAUNT)
        HCPTKDEF 'UNCLE',NEXT=ST_3,
            ORFLAG=(FROM,FUNCLE)
        HCPTKDEF TYPE=TOKEN,NEXT=ST_4,    For 'friend'
            CONFLICT=1,                  Do not ask for money
            ORFLAG=(FROM,FFRIEND),
            STORE=CWHO

ST_2     HCPSTDEF REQMATCH=YES
        HCPTKDEF TYPE=TOKEN,NEXT=ST_4,    For 'name'
            STORE=CWHO

ST_3     HCPSTDEF REQMATCH=YES
```

	HCPTKDEF TYPE=TOKEN,NEXT=ST_4, STORE=CWHO	For 'name'
ST_4	HCPSTDEF REQMATCH=YES HCPTKDEF 'FOR'	
ST_5	HCPSTDEF REQMATCH=NO,NEXT=ST_7 HCPTKDEF TYPE=DEC,NEXT=ST_6, CONFLICT=1, RANGE=(1,10000), STORE=HOWMUCH	For 'n' Do not ask friend No more than 10,000
ST_6	HCPSTDEF REQMATCH=YES,MISSING=MS670401 HCPTKDEF 'DOLLARS',NEXT=ST_9, ORFLAG=(WHAT,FUSD)	
ST_7	HCPSTDEF REQMATCH=NO,NEXT=ST_8 HCPTKDEF 'THE',NEXT=ST_8	
ST_8	HCPSTDEF REQMATCH=YES HCPTKDEF 'CAR',NEXT=ST_9, ORFLAG=(WHAT,FCAR)	
ST_9	HCPSTDEF REQMATCH=NO,FINISH=YES HCPTKDEF TYPE=NULL	

Step 8: Develop the HCPDOSYN Macro

The HCPCFDEF, HCPSTDEF, and HCPTKDEF macros have not built any control blocks to define the structure. They only updated macro variables needed to define the structure. The HCPDOSYN macro will build the necessary control blocks that the parser will use.

As we mentioned earlier, we need to inform the parser about the data areas that we will use, parameter lists that we use, and inform the parser about which messages to generate should we find any one of a number of miscellaneous errors.

Because we are storing into one data area, we can use this as our primary parameter list. In our case, we will code the HCPDOSYN macro with the ROOT=YES operand to indicate we want pointers to the various pieces of syntax and any error message information common to all the pieces, the PLBASE=ASKDSECT operand to indicate the name of the primary plist and the PLEN=LENDSECT operand to indicate the length of the plist in bytes. We will also code the CONFLICT=MS001301 operand to indicate which error message should be generated when a conflicting option is detected.

CFDEF	HCPCFDEF 'ASK',PROCESS=NONE	
ST_1	HCPSTDEF REQMATCH=YES HCPTKDEF 'DAD',NEXT=ST_4, ORFLAG=(FROM,FDAD) HCPTKDEF 'MOM',NEXT=ST_4, ORFLAG=(FROM,FMOM) HCPTKDEF 'AUNT',NEXT=ST_2, ORFLAG=(FROM,FAUNT) HCPTKDEF 'UNCLE',NEXT=ST_3, ORFLAG=(FROM,FUNCLE) HCPTKDEF TYPE=TOKEN,NEXT=ST_4, CONFLICT=1, ORFLAG=(FROM,FFRIEND), STORE=CWHO	For 'friend' Do not ask for money
ST_2	HCPSTDEF REQMATCH=YES HCPTKDEF TYPE=TOKEN,NEXT=ST_4, STORE=CWHO	For 'name'
ST_3	HCPSTDEF REQMATCH=YES HCPTKDEF TYPE=TOKEN,NEXT=ST_4, STORE=CWHO	For 'name'
ST_4	HCPSTDEF REQMATCH=YES HCPTKDEF 'FOR'	
ST_5	HCPSTDEF REQMATCH=NO,NEXT=ST_7 HCPTKDEF TYPE=DEC,NEXT=ST_6, CONFLICT=1, RANGE=(1,10000), STORE=HOWMUCH	For 'n' Do not ask friend No more than 10,000

```

ST_6    HCPSTDEF REQMATCH=YES,MISSING=MS670401
        HCPTKDEF 'DOLLARS',NEXT=ST_9,
            ORFLAG=(WHAT,FUSD)

ST_7    HCPSTDEF REQMATCH=NO,NEXT=ST_8
        HCPTKDEF 'THE',NEXT=ST_8

ST_8    HCPSTDEF REQMATCH=YES
        HCPTKDEF 'CAR',NEXT=ST_9,
            ORFLAG=(WHAT,FCAR)

ST_9    HCPSTDEF REQMATCH=NO,FINISH=YES
        HCPTKDEF TYPE=NULL

DOSYN   HCPDOSYN ROOT=YES,
        PLBASE=ASKDSECT,
        PLEN=LENDSECT,
        CONFLCT=MS001301

```

We have just finished defining the syntax table to handle our new command.

Step 9: Write the Code to Call the Parser

In this example, we are writing a new command so we will have to write code in the command handler to pass the command to the parser and act on the response from the parser. When coding the invocation of the parser, you should remember to pass an address, where appropriate, or zero.

HCPGETST LEN=(R0)	Get ASKDSECT area
LA R0,LENDSECT	*
LR Rx,R1	Base register
HCPUSING ASKDSECT,Rx	*
L R0,=A(DOSYN)	Address of HCPDOSYN
LA R1,ASKDSECT	Address of answer plist
LA R2,0	Address of add'l bases
L R3,=A(CFDEF)	Address of HCPCFDEF
HPCALL HCPZPRPC	Parse the command line
ST R0,SAVER2	Store returned msg number
LTR R15,R15	Check out return code
BNZ EXIT	Msg already issued by HCPZPR
TM WHO,FDAD	Was it ASK DAD?
BO ASKDAD	..yes
TM WHO,FMOM	Was it ASK MOM?
BO ASKMOM	..yes
:	
LA R1,ASKDSECT	Release gotten storage
HCPRELST BLOCK=(R1)	*
HCPDROP Rx	Drop base register

For more information on the parser routines and their calling conventions, see [Appendix G, “CP Parser Macros and Routines,”](#) on page 189.

Step 10: Write the Commands to Activate the Code

We will leave this step to you because it is discussed in other areas of this document. You will need to write at least CPXLOAD statements and DEFINE statements for the SYSTEM CONFIG file.

Using the Parser to Store Data

The parser provides a mechanism to convert some external data, like a command or a configuration file statement, into an internal format, like a fixed binary 32 bit number. To make this work, you need to provide the parser two basic pieces of information:

1. the rules for performing the conversions
2. the place or places to store the results.

The rules for performing the conversions are called the syntax and you define these rules using the syntax definition macros (i.e. HPCFDEF, HCPSTDEF, HCPTKDEF, HCPDOSYN). The places to store the results are

called the parameter lists (or plists). You code your syntax definition macros to identify the plists that are used and to indicate how data is to be stored in these plists.

The parser uses two types of plists:

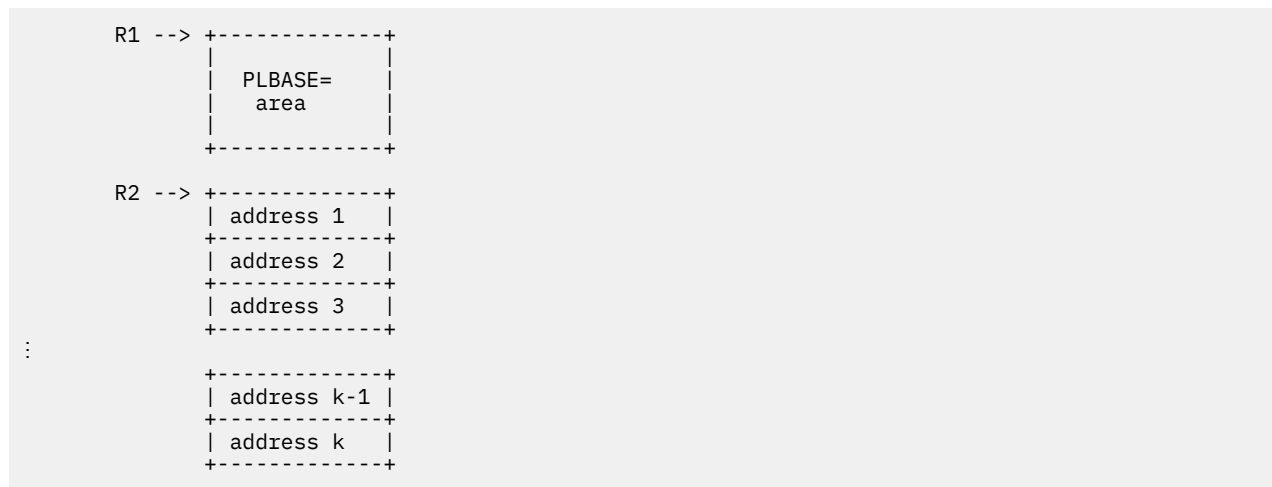
1. Specified by PLBASE= on the HCPDOSYN macro

The plist specified by PLBASE= is known as the primary plist. It is also known as plist number 0. You are allowed to specify one primary plist. The parser will clear the storage associated with this plist when it starts parsing the command or configuration file statement.

2. Specified by BASES= on the HCPDOSYN macro

The plists specified by BASES= are known as the secondary plists. You can have none or several of these depending on your needs. These represent other areas where you might want to store data. They are numbered starting from 1. The plists specified by BASES= are never cleared.

Here is a picture of how the plists get passed to the parser:



It is important that you build the list of addresses that R2 points correctly. That is, the content and the order must match what you have defined for the syntax description with the BASES= parameter of the HCPDOSYN macro. If your syntax definition does not specify any secondary plists with BASES=, then you would pass R2=0 to the parser.

The syntax will tell the parser which plist to use to store the results of the conversions. This is done (that is, the parser is told) by means of the HCPTKDEF and the HCPDOSYN macros.

Storage locations typically are specified in the parser syntax macros by these methods:

```
keyword=where
keyword=(where,what)
```

Examples would be:

```
STORE=where
ORFLAG=(where,what)
ANDFLAG=(where,what)
```

The value of "what" is simply anything that would be acceptable in an Or-Immediate (OI) instruction or in an And-Immediate (NI) instruction. The value of "where" typically is specified in either of these 2 forms

```
field
base(field)
```

A simple "field" specification is always assumed to be based off the primary plist (that is, the PLBASE= value you specify on the HCPDOSYN macro). A "base(field)" specification is based off the secondary plist (that is, the BASES= value you specify on the HCPDOSYN macro).

HCPTKDEF saves in assembler variables (for use later by the HCPDOSYN macro) the names of the DSECTs into which data are to be stored. For example, these example keywords

```
ORFLAG=(SYSCM(SYSIPLFL),SYSAUTOW)
STORE=SYSCM(SYSCKVOL)
STORE=SCFMACH
```

would cause HCPTKDEF to remember the target locations as

```
SYSCM(SYSIPLFL)
SYSCM(SYSCKVOL)
SCFMACH
```

Later, HCPDOSYN will look at each target location. HCPDOSYN first tries to match the target location to something in BASES=. It does this by separating the string into what appears to be a DSECT and a label (in these examples, SYSCM and SYSIPLFL; SYSCM and SYSCKVOL). If it cannot perform the separation, then the target location must be in the primary plist (in this example, SCFMACH must be in the primary plist).

After breaking apart the target location, HCPDOSYN searches for the apparent DSECT name in the BASES= list. If it does not find it, then the target location must be in the primary plist. The original target location string will be used. If it does find it, then it remembers which entry in BASES= satisfied the search. Finally, HCPDOSYN generates the syntax control blocks, where it saves the number 'n' of the plist area ('n' = 0 for the primary plist, 'n' > 0 for something in the BASES= list) and the offset into that DSECT to the target location.

Eventually your system is IPLed and the parser is called. One of the parameters passed to the parser is R2, which has the address of the list of addresses that you set up to match the list of DSECT names in the BASES= keyword. As the parser processes the input against the syntax, it finds that something should be stored somewhere and looks at the plist number that HCPDOSYN saved.

If the plist number 'n' = 0, then the parser uses whatever was in R1 as the base address of the primary plist. Adding the offset to that value gives the address of the field.

If the plist number 'n' > 0, then the parser uses whatever was in R2 to be the address of a list of pointers, takes the n-th pointer as the base address of the secondary plist. Adding the offset to that value gives the address of the field.

In the examples above, if we had this on our HCPDOSYN

```
PLBASE=SCFBK
BASES=(PFXPG,RDEV,SYSCM)
```

then HCPDOSYN would remember that SYSCM should be the third pointer in the list. Any reference to SYSCM in the syntax macros would be converted to plist position '3'. When the parser executes, it will take this plist number '3' and use the third address located by R2 as the base address for SYSCM. The list of addresses that you build and that point to with R2 should have the address of SYSCM in the third position. If not, then the parser will get the wrong address and store unpredictably.

If we call the parser this way (assume that at runtime that MYBASES has been filled in with addresses that the statements suggest and that you have obtained storage for the primary plist you have selected which is SCFBK in this example):

```

      L      R0,...
      LA     R1,SCFBK           Address of primary plist
      LA     R2,MYBASES        Address of secondary plists
      L      R3,...
      HPCALL the parser
:
MYBASES DSECT
        DS   A(PFXPG)
        DS   A(RDEV)
        DS   A(SYSCM)
```

then the plist numbers that HCPDOSYN saved are in the same sequence as the control block addresses in the list that R2 points to, and good things should result.

If, however, we call the parser this way (addresses of SYSCM and PFXPG are switched around in the MYBASES DSECT), bad things will happen.

```

L      R0,...
LA     R1,SCFBK      Address of primary plist
LA     R2,MYBASES    Address of secondary plists
L      R3,...
HCPCALL the parser
:
MYBASES DSECT
DS      A(SYSCM)
DS      A(RDEV)
DS      A(PFXPG)

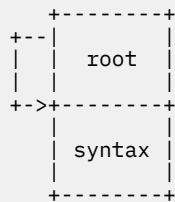
```

The parser will try for the third pointer in MYBASES when it needs the address of SYSCM (because HCPDOSYN remembered 'n' = 3 as the plist number for the SYSCM DSECT) and bad things will happen. Where we want to store into offset X'6D3' into SYSCM for SYSIPLFL, we will actually be storing into PFXPG plus offset X'6D3'.

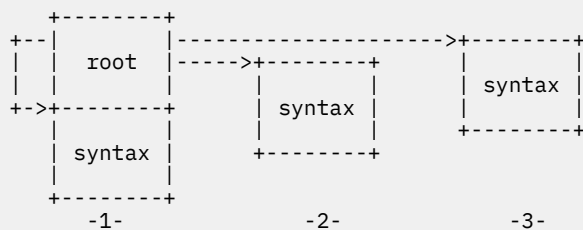
Dividing Your Syntax over Multiple Modules

The syntax you define can contain many statements of varying complexity. It is possible that a syntax could grow too large to be conveniently contained in a single module. In this case, you would need to define pieces of the syntax in separate modules and then connect them so they can be used together. To do so, you would need to provide a main or "root" piece that would contain pointers to the various pieces of the syntax. The "root" piece would also contain information (such as error message equates) that would be common for all pieces of the syntax.

In a single syntax piece, both the "root" and the syntax itself are generated together automatically.



In a situation where you have pieces of your syntax spanning multiple modules, you only want one "root" defined.



To generate the "root", use ROOT=YES on the HCPDOSYN macro. To avoid the "root", use ROOT=NO on the HCPDOSYN macro. To connect the "root" to the other two pieces of the syntax, use the SYNLIST= keyword on the HCPDOSYN macro.

For this picture, you would have three modules, each would define its piece of the syntax and would have its own HCPDOSYN macro. The SYNLIST= option on the root HCPDOSYN macro is used to tie the various syntax pieces together.

For the first module, you would have:

```
HCPDOSYN ROOT=YES,SYNLIST=(hcpdosyn2,hcpdosyn3)
```

We use the notation "hcpdosyn2" in order to indicate that you would specify the label on the HCPDOSYN macro in module 2. Similarly, the notation "hcpdosyn3" indicates that you would specify the label on the HCPDOSYN macro in module 3.

For the second module, you would have:

```
HCPDOSYN ROOT=NO
```

For the third module, you would have:

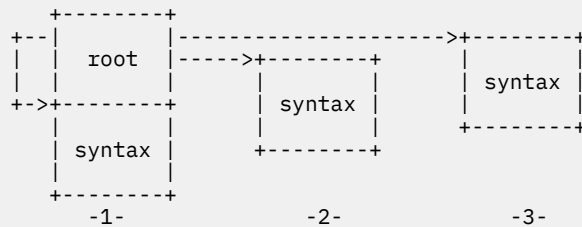
```
HCPDOSYN ROOT=NO
```

You should note that the HCPDOSYN macro generates an HCPENTRY instruction for the labels that it generates so that they can be referenced externally from the module which contains the HCPDOSYN macro. If you were to specify your own label on the HCPDOSYN macro, then you would have to specify your own HCPENTRY instruction.

Combining ROOT=, PLBASE=, BASES=, and SYNLIST=

In the case of multiple syntax pieces, each must have the same set of plists defined. The parser is given only one set of addresses (in R1 and in R2). If the addresses do not match the labels on the PLBASE= and the BASES= keywords, then the parser will use the wrong address when storing the result of its parsing.

So, in the case where we have multiple syntax pieces, we must specify identical plists for all three pieces of the syntax.



For the first module, you would have:

```
HCPDOSYN ROOT=YES,SYNLIST=(hcpdosyn2,hcpdosyn3),
          PLBASE=dsect0,BASES=(dsect1,dsect2)
```

For the second module, you would have:

```
HCPDOSYN ROOT=NO,
          PLBASE=dsect0,BASES=(dsect1,dsect2)
```

For the third module, you would have:

```
HCPDOSYN ROOT=NO,
          PLBASE=dsect0,BASES=(dsect1,dsect2)
```

Using a Post-Processor

There are times when the parser will be able to handle all the processing you require for your new command or configuration file statement. For example, if the only thing you need to do is to save some information into existing control blocks, and the parser has access to these control blocks, then you can let the parser do that work for you.

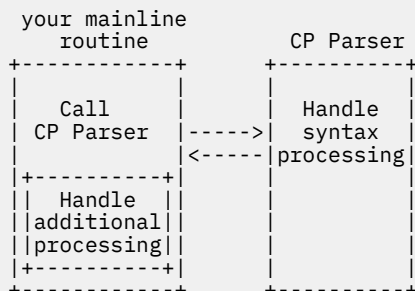
If, however, you have additional processing that you would like to perform after the parser has processed the syntax, then you have two choices. You can either

1. handle that additional processing in your mainline routine, or
2. you can move that processing to another routine that you would identify as your post-processor.

Let us look at an example of each.

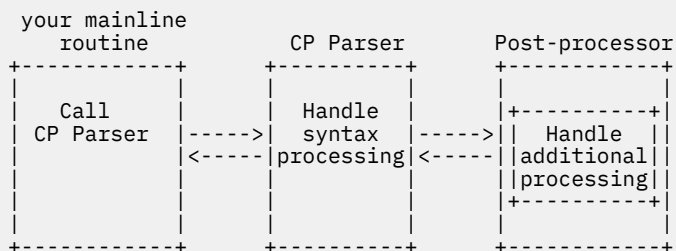
Providing Additional Processing in Your Mainline Routine

If you have used the syntax macros to define the syntax for your own command, any additional processing that you want to perform can be coded in your mainline routine. This is, in general, the easiest thing to do. For example, consider the ASK command developed in previous examples. You did not need to use a post-processing routine because all additional processing was handled in the mainline routine you wrote. That is, you wrote code in your command handler (your mainline routine) to pass the command to the parser and to act on the results of the parsing.



Providing Additional Processing in Your Post-processor.

You could have chosen to use a post-processor routine instead of handling all the additional processing in your mainline routine. The post-processor routine is specified on the `PROCESS=` keyword of the `HCPCFDEF` macro. If you specify the name of a routine, that routine will be called by the parser as long as the parser processing completes without detecting an error. The post-processor performs its own processing, and will return to the parser, which will finally return to the mainline routine that called the parser.



Typically, you only need to consider using a post-processor if you have used the syntax macros to define the syntax for your own configuration file statement. For configuration file statement processing, you do not provide the mainline routine that calls the parser. Rather, CP does the work of reading the configuration file to isolate statements and CP does the work of setting up the calls to the parser. Since you do not have a mainline routine of your own where you can code any additional processing, if you have additional processing you wish to perform, you must provide that additional processing in a post-processing routine.

Using a Single Syntax for a Command and for a Configuration File Statement

If you are developing something that you intend to use both as a command and as a configuration file statement, you need to code for the mixed environment.

- Since the syntax macros will be used for parsing the configuration file statement, you will probably code a post-processor.
- Since these same syntax macros will be used for parsing the command, the same post-processor will be used (because the same `HCPCFDEF` macro will be used by the parser, and that `HCPCFDEF` macro specifies the post-processor name).

Therefore, all additional processing that you want to do will have to be done in the post-processor, whether for the statement or for the command.

Creating Your Own Configuration File Statements

The root syntax definition for all of CP's configuration file statements is found in HCPZSC. The EXTERNAL_SYNTAX statement provides a way to identify a syntax definition that you want to have included as part of the syntax table for CP's configuration file statements. CP will do the work of joining your syntax to the other syntax pieces so that you do not have to make any updates to the existing syntax definitions in HCPZSC.

To establish your own configuration file statement, you need to define the syntax using the syntax definition macros (HPCFDEF, HCPSTDEF, HCPTKDEF, HCPDOSYN). In addition, you optionally provide a post-processor subroutine to be driven if the parsing completes successfully. You only need a post-processor if there is additional checking or processing that you would like to do after all parser processing is completed.

The EXTERNAL_SYNTAX statement you specify in the configuration file identifies the syntax definitions you have coded for your configuration file statements. CP will treat that syntax definition as an additional piece of the existing syntax in HCPZSC, as if you had coded a SYNLIST= on the root HCPDOSYN macro invocation. CP will insert your syntax definition into the syntax table so that it is examined and used before any of the syntax definitions that CP has for its own configuration file statements. Each EXTERNAL_SYNTAX syntax will be inserted into the syntax table so that it is examined and used before any other existing syntax definitions. EXTERNAL_SYNTAX gives you the ability to add your own configuration file statement or to replace any existing CP statement.

Because you are defining a piece of a syntax to be added to the existing syntax tables for configuration file statements, there are a few coding considerations to keep in mind.

- Primary Plist (PLBASE=)

The PLBASE= parameter on the root HCPDOSYN macro in HCPZSC defines the SCFBK as the primary plist that all configuration file statements use for parsing. This means that your syntax definition must also specify SCFBK as the primary plist on the PLBASE= parameter of your HCPDOSYN macro. You cannot specify anything else.

If you plan to store information into the primary plist, the SCFMISC field is the area in that control block that you can redefine and use for your own purposes. Your syntax definition and your post-processor do not necessarily have to use the primary plist. For example, all of the storing you ask the parser to do may involve only the data areas specified in the secondary plists.

If you redefine the SCFMISC field, make sure that you do not exceed its size. You can make sure of this by using the following coding technique.

SCFBK	DSECT	,	Return to SCFBK
	ORG	SCFMISC	Redefine SCFMISC
MYFIELD1	DS	...	One of my fields
MYFIELD2	DS	...	Another of my fields
	CKMAINT	(*-SCFMISC),GT,(L' SCFMISC)	Error if true
	ORG	,	Return to end of SCFBK

- Secondary Plists (BASES=)

The BASES= parameter on the root HCPDOSYN macro in HCPZSC defines the secondary plists that all configuration file statements use for parsing. This is the list of bases that are used by the syntax macros to figure out where they have to store information. Because all of the syntax definitions for all of the configuration file statements are considered part of the same syntax structure, you must code the identical list on the BASES= parameter of your HCPDOSYN macro. Your syntax may not necessarily make use of each entry in the bases list, however, your syntax definition must have that list in the same order as HCPZSC. You cannot change the order, delete any entries, or add new control blocks to the list. If you want to save some information in a control block that is not listed in BASES=, then this is work you must defer to your post-processor. For example, your syntax could save information in the primary plist and then your post-processor routine could move that information to a different location.

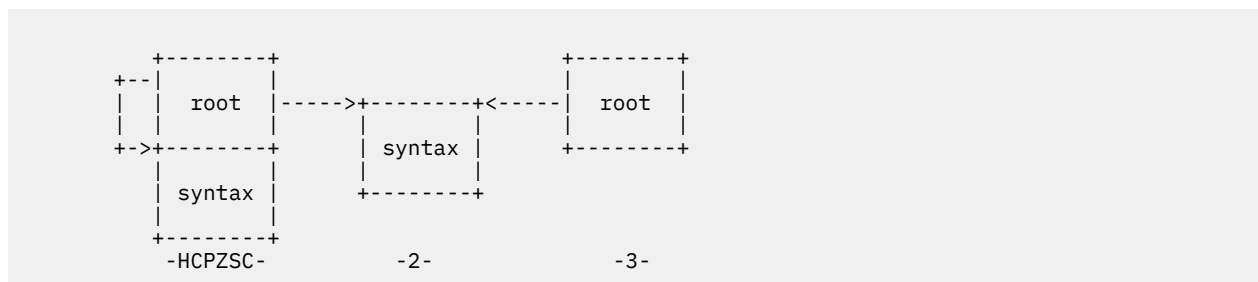
The file HCPZSCBS COPY maps the pointers required by the syntax definition in HCPZSC for configuration file statements. You can use this to interpret the list of base addresses that are passed to your post-processor routine.

- Specifying ROOT=

The main control block that CP uses to find all of the syntax related control blocks is created as a result of the ROOT=YES parameter on the HCPDOSYN macro. Just as with any syntax definition that spans multiple modules, only one of the syntax pieces can be considered the root. Your syntax definition is considered a secondary syntax piece, so the syntax definitions that you use must have ROOT=NO specified on their HCPDOSYN macros.

Using One Syntax for Two Purposes

It is possible to connect the same syntax for more than one use. By this, we mean that more than one "root" would connect to the same syntax.



This is typically done in the situation where the syntax describes something you intend to use as both a configuration file statement and as a CP command. Let's take this picture and show how the various pieces of syntax would be tied together and how the HCPDOSYN macro should be coded for each.

In this example the first module represents HCPZSC which contains the root syntax definition for all of CP's configuration file statements. The second module would contain the syntax you have defined. The third module would be your command handler which will call the parser to parse your command.

As we have explained before, the PLBASE= and the BASES= parameters must be the same for all pieces of a syntax that are joined together so that the parser will be able to determine correct storage locations. Because of the coding rules for configuration file statements with regard to the plists, then, the PLBASE= and BASES= parameters in this example will all have to match the ones which HCPZSC defines for all configuration file statements.

The HCPDOSYN macro invocation in HCPZSC (the first module in our example) defines the root syntax for all configuration file statements. The SYNLIST= entries would identify other pieces of the syntax found in various CP modules that comprise the configuration file syntax table. The PLBASE= and BASES= keywords would identify the plists that are used. For the purposes of this example, let's assume the HCPDOSYN macro invocation in HCPZSC is as follows:

```
HCPDOSYN ROOT=YES,
          SYNLIST=(HCP...,
:
          HCP...),
          PLBASE=SCFBK,
          BASES=(aaa,
          bbb,
          ccc)
```

For the second module which defines your syntax, notice that the value for PLBASE= and BASES= both match what is specified for HCPZSC above. In addition, the ROOT=NO keyword is used because this is just a piece of a syntax:

```
HCPDOSYN ROOT=NO,
          PLBASE=SCFBK,
          BASES=(aaa,
          bbb,
          ccc)
```

To make the connection between the syntax table in HCPZSC and your syntax definition in module 2, you would code an EXTERNAL_SYNTAX statement like the following:

```
EXTERNAL_SYNTAX  EPNAME  hcpdosyn2
```

The notation "hcpdosyn2" represents the label on the HCPDOSYN macro invocation in module 2.

For the third module, notice that the value for PLBASE= and BASES= both match what is specified in module 2. The ROOT=YES keyword is needed because this will be the root of the syntax used for command processing. The SYNLIST= keyword connects the root to the syntax piece you defined in module 2:

```
HCPDOSYN ROOT=YES,
          SYNLIST=(hcpdosyn2),
          PLBASE=SCFBK,
          BASES=(aaa,
                bbb,
                ccc)
```

Error Messages and How the Parser Handles Them

The parser can issue error messages of your choice for most of the syntax errors that it detects. Message numbers for invalid command verbs, invalid keywords, conflicting keywords, extraneous keywords, and so forth are specified on the HCPDOSYN macro invocation. If there is no message number on the HCPDOSYN macro for a detected condition, the parser does not issue a message when it detects the invalid syntax, but a return code reflects the invalid syntax to the calling routine.

If a parsed token does not match any keywords specified on one of the HCPTKDEF macros under the current HCPSTDEF and the HCPSTDEF is coded with REQMATCH=YES, the parser issues a bad keyword message from the BADKWD option on the HCPDOSYN macro.

If the last HCPTKDEF in the list contains only a conversion type and does not have a keyword, and the token is invalid, the ERROR specification on the HCPTKDEF macro determines the error message to be issued. If the ERROR specification is omitted, the parser uses the BADKWD specification from the HCPDOSYN macro instead.

If the parsed token matches a keyword specified on a HCPTKDEF entry for the current HCPSTDEF, and the HCPTKDEF entry also contains a TYPE specification (thus requiring an additional token), one of two error conditions may occur:

1. There may not be an additional token. In this case, the parser issues the error message specified on the MISSING parameter of the HCPSTDEF macro or of the HCPDOSYN macro.
2. The additional token may be invalid because of the TYPE specified on the HCPTKDEF macro. In this case, the parser uses the message number specified on the ERROR parameter of the HCPTKDEF macro. If you omit the ERROR parameter, the parser uses the BADKWD message number from the HCPDOSYN macro instead.

When the parser is called upon to issue error messages before the end of the IPL process, it assumes the error messages come from the processing of configuration files read by CP. During this phase, the parser builds all error messages by invoking the HCPCONSL macro with the DESTINATION=GSDBK option, thus causing the message builder to place the message in a general system data block (GSDBK). The GSDBK is then placed on a queue of GSDBKs anchored at HCPISUSC. This queue of GSDBKs is later issued to the operator console when the location of the operator console has been determined.

Once the system is up and running, the parser issues any required messages in the form of EMSGs back to the console of the virtual machine definition block (VMDBK) it is running under. Thus, when the parser detects an error in a command a user has issued, it automatically sends the error message back to the user's console.

For errors encountered in reading configuration files, the parser prefaces the actual error message with an indication of where in the file the error occurred. This message includes the file name, file type, and record number (or numbers) where the statement was.

Detecting Syntax Errors

The parser will not commit changes to data areas if an error is detected while parsing the command or statement. If the parser were to process an input string from left to right and stop when it encountered a parsing error, it would be inconvenient if any changes to data areas had been made. For example, consider the CHARACTER_DEFAULTS statement:

```
*
*-----*
* CHARACTER_DEFAULTS statement *
*-----*
*
CFCHAR   HCPCFDEF 'CHARACTER_DEFAULTS',PROCESS=NONE
*
CHARLOOP HCPSTDEF REQMATCH=YES,VALEND=YES,MATCH1=YES,NEXT=CHARLOOP
          HCPTKDEF 'LINE_END',TYPE=CHAR,STORE=SYSCM(SYSLEND),
              CONFLICT=1,ERROR=MS670605
          HCPTKDEF 'LINE_DELETE',TYPE=CHAR,STORE=SYSCM(SYSLDEL),
              CONFLICT=2,ERROR=MS670605
          HCPTKDEF 'CHAR_DELETE',TYPE=CHAR,STORE=SYSCM(SYSCDEL),
              CONFLICT=3,ERROR=MS670605
          HCPTKDEF 'ESCAPE',TYPE=CHAR,STORE=SYSCM(SYSESCP),           X
              CONFLICT=4,ERROR=MS670605
          HCPTKDEF 'TAB',TYPE=CHAR,STORE=SYSCM(SYSTAB),               X
              CONFLICT=5,ERROR=MS670605
```

If the parser updated the SYSCM area directly as it parsed from left to right, then a statement such as the following would cause the parser to update the SYSLEND, SYSLDEL, SYSCDEL, and SYSESCP fields in the system common area before it encountered the invalid TAB character specification of *arf*:

```
Character_Defaults      ,
  Line_End              #      , /* Use hash mark as line end character */
  Line_Del              Off    , /* Do not have a logical line del char */
  Char_Delete           @      , /* Use at sign as logical char del char */
  Escape                '""'   , /* Use double quote as escape character */
  Tab                   arf     , /* Use this as a tab character */
```

Because any invalid syntax invalidates the whole statement, the changes to SYSCM would have to be replaced by the original values.

Instead of taking this approach, the parser saves up the changes caused by parsing the statement and only applies them to the actual data areas when the entire statement has been parsed and no syntax error has been detected. This would occur before invoking a post-processor. Using this approach with the above example, the parser would not change SYSCM, and it would issue message 6706 version 5 for the *arf* token.

Note: The only data that is *not* saved up until the parser has finished checking the syntax is the accumulation requested by the ACCUM parameter. The fields referred to by this parameter must reside in the primary output data area identified by the PLBASE parameter on the HCPDOSYN macro.

Additional Features

Another feature of the parser is the way it handles tokens that could satisfy more than one HCPTKDEF in a given state. If, for instance, you have a command that accepts either a device address or a volume identifier in a particular position, you can code HCPTKDEF macros as follows:

```
HCPSTDEF ...
HCPTKDEF TYPE=HEX,RANGE=(0,X'FFFF'),STORE=DEVADDR
HCPTKDEF TYPE=TOKEN,STORE=DEVVOLID
```

In such a case, if the token parsed from the command line turns out to be a valid device address, the parser stores the binary equivalent of the address in the DEVADDR field. If the token is not a valid device address and is less than 6 characters long (the defined length of the DEVVOLID field), it is uppercased and stored in the DEVVOLID field. The post-processing routine can then check whether the DEVVOLID field contains binary zeros to determine whether a device address or a volume identifier was specified on the command.

Chapter 10. Common and Frequent Problems

This section discusses many of the problems that can occur when you try to customize z/VM using dynamically loaded routines. Unfortunately, it is not possible for us to anticipate all of the problems that you may encounter. Instead, we will try to cover the most common and most frequent mistakes.

This section is divided into two topics that:

- Discuss the diagnostic facilities available in handling problems related to CP exit points
- List the common mistakes, how to recognize them, and how to solve them.

Available Diagnostic Facilities

We have attempted to provide new commands, ensure that existing commands were adequate and add information where necessary to assist you in diagnosing problems.

Before we begin discussing the diagnostic tools, we should mention the benefits of diagnosing problems on a second-level test system. When you test a second level system you can view the code flow in a step-by-step manner. There are numerous commands that can be issued to show locations of control blocks and routines. These addresses will be useful in setting trace points to monitor the code flow.

The following commands were added specifically for diagnosing problems related to CP Exits:

QUERY CPCMDS

Returns information about the various versions of a command. This information includes: entry point name of the command handler, current privilege classes associated with the command version, whether it is enabled or disabled, and information about the primary command (if the command being queried is an alias).

QUERY DIAGNOSE

Returns information about a Diagnose code. This information includes: entry point name of the Diagnose code handler, assigned privilege classes and whether the Diagnose code is enabled or disabled.

QUERY CPXLOAD

Returns information about loaded routines. This information includes: options entered on the CPXLOAD command (does not include any CPXLOAD directives imbedded in the loaded files). Also included are: CSECT names, address and sizes, entry point names and addresses, name of the loaded file, and the time and date of the load.

QUERY EXITS

Returns information about the specified exits. This information includes: enabled/disabled status, entry point names associated with the exit, and total number of calls (successful and attempted) and total elapsed time spent in the entry point.

QUERY UNRESOLVED

Returns information about any unresolved references as the result of a CPXLOAD. This information includes the names of CSECTs that refer to an unresolved entity (entry point/symbol) and the name of the unresolved entity.

LOCATE CMDBK

Returns the address of the CMDBK for a specific command.

LOCATE DGNBK

Returns the address of the DGNBK for a specific Diagnose code.

LOCATE ICLBK

Returns the address of the CP indirect call locator block for a specific entry point.

LOCATE XITBK

Displays the address of the CP exit block for a specific entry point.

QUERY CPLANGLIST

Returns information about languages in which CP can display messages to the specifier's virtual machine. Also returns the message repository languages, their component IDs, and external symbols.

QUERY ICLNAME

Returns usage statistics information about indirect calls to external entry points. This information includes the number of times the entity was successfully called and the total elapsed time (not CPU time) spent in the entry point.

There are many other CP commands (such as DISPLAY) which you will find useful. For more information about the command syntax and use of the commands see [z/VM: CP Commands and Utilities Reference](#). For additional information on debugging CP, see [z/VM: Diagnosis Guide](#).

Viewing a CP trace table may become necessary when you are diagnosing a problem. CP Exit processing will create trace records to show the code flow into and out of exits. The exit codes specific to exit processing are:

F900

Call Exit Start identifies the beginning of processing for a specific exit point.

F910

Exit Routine Start identifies the beginning of processing of an exit routine associated with a specific entry point.

F920

Exit Routine Finish identifies the end of processing of an exit routine associated with a specific entry point.

F930

Call Exit Finish identifies the end of exit point processing for a specific exit point.

F940

Call Exit Routine Finish (NOP) identifies the inability to call an exit routine during exit point processing. This can occur if the routine is no longer loaded.

2810

Indirect Call Request identifies an indirect call to a routine.

2C10

Indirect Call Return identifies the return from a routine that was indirectly called.

The following shows a set of trace entries you might see when viewing the CP trace table:

```
F900 ... Exit 1200
F910 ... Exit 1200
2810 ...
2C10 ...
F920 ... Exit 1200    CtlRc=0 MainRc=4
F910 ... Exit 1200
F940 ... Exit 1200
F930 ... Exit 1200    MainRc=4
```

In the previous example, exit 1200 is invoked. It in turn called an exit routine and gave control to it indirectly. After the first exit routine was called an attempt to invoke a second exit routine was made but that invocation failed because the routine was no longer callable. The final return code from the exit point was four.

A bit of defensive programming that will aid diagnosis or prevent the need for diagnosis is the CKMAINT macro, Check Future Maintenance macro. This macro provides a way to automatically check that future maintenance does not, without knowing it, change a value that could affect the reliability of the system. This macro is used during compilation to ensure that generated sizes such as a table size remain addressable or usable. For example, you may create a data area that is less than 256 bytes long so that you can use an MVC, move character instruction, to change the whole area. At the time you develop the original code, you would be wise to code a CKMAINT to verify that the data area is never unwittingly changed by later developers to a size greater than 256.

Finally, you may encounter times, usually in diagnosing a lock related problem, when you will need to cause an abend so that you can attain more data on the problem. We recommend that you try to cause

and recreate these types of problems second level because forcing an abend is what we consider to be the final diagnostic choice. Users do not enjoy having the system go suddenly unavailable in order to take a system dump.

However, if you have to take a system dump, the HCPASERT macro will be very useful. This macro allows a programmer to declare that certain conditions must be true at a given point in the code. This macro will (essentially) generate code that checks the assertion. If the assertion is not true, an abend is generated (by default). In addition, to verifying that specified locks are held, you can also specify other assertions such as the code is running on the master processor, or the VMDBK is dispatched.

When CP is running as a second level system, HCPASERT is ideal for avoiding dumps related to an assertion problem. HCPASERT processing can indicate which assertion failed and drop you into CP READ allowing you to further debug the problem.

Note: The assertion checking performed by HCPASERT is activated by the SET CPCHECKING command or using the FEATURES configuration file statement.

The HCPAIF, HCPAELSE, and HCPAEND macroinstructions can be used to define alternate instruction sequences. HCPAIF, HCPAELSE, and HCPAEND are structured programming alternatives to the Assembler Language AIF and AGO pseudo-instructions and statement sequence symbol labels. The operand of HCPAIF must be a simple boolean value; expressions are not permitted. The operands of HCPAELSE and HCPAEND are comments and are permitted in order to improve readability. Ordinarily, excluded blocks of statements are shown in the listing even though no object code is generated for them. This facilitates code review processes. HCPAIF, HCPAELSE, and HCPAEND may not be used in macros.

Common Mistakes

This section discusses the common mistakes that can be made when customizing CP. It is divided into a series of topics (for example, problems with defining new commands). Each topic discusses the common mistakes that can occur and the diagnostic functions that you can perform to identify the problem. Suggested readings for further information are provided for each topic.

The topics include:

- [“Command Related Problems” on page 83](#)
- [“Diagnose Code Related Problems” on page 84](#)
- [“Message Related Problems” on page 85](#)
- [“Exit Point Related Problems” on page 85](#)
- [“CPXLOAD Related Problems” on page 86](#)
- [“CPUNXLOAD Related Problems” on page 87](#)
- [“Miscellaneous CP Customization Errors” on page 87](#)

Command Related Problems

- **Symptom: My command is not being recognized.**

Some of the causes of this symptom are:

- The necessary versions of the command are not defined or enabled.
- The command is an alias and both the alias and its base command are not enabled.
- The necessary privilege classes have not been assigned to the command.
- The entry point for the command handler is not loaded.
- The wrong entry point name was specified on DEFINE COMMAND or MODIFY COMMAND.

The following commands will be useful:

- QUERY CPCMDS *command*
- LOCATE SYMBOL *entry_point_name*
- QUERY ICLNAME *entry_point_name*

- QUERY CPXLOAD EPNAME *entry_point_name*

- **Symptom: The system abends when my command is invoked.**

Some of the causes of this symptom are:

- An address constant in your dynamically loaded routine was not resolved. Note: QUERY UNRESOLVED will show the unresolved symbol references in a routine that was dynamically loaded using CPXLOAD. It will not show unresolved symbol references that resulted during a build of the system.
- The necessary calling parameters were not defined for your entry point (e.g. MP specified instead of NONMP).
- The EPNAME value specified on the DEFINE COMMAND or the MODIFY COMMAND is an external symbol (for example, the CSECT name of your routine) rather than the entry point name which is coded to set up your routine's addressability.
- Your routine is not compiled with the correct level of the maclibs.

The following commands will be useful:

- QUERY UNRESOLVED
- LOCATE SYMBOL *entry_point_name*

Additional Readings

For additional information on defining new commands, see [Chapter 6, “Defining and Modifying Commands and Diagnose Codes,”](#) on page 43.

Diagnose Code Related Problems

- **Symptom: My Diagnose code is not being recognized.**

Some of the causes of this symptom are:

- The diagnose code is not enabled.
- The correct privilege classes were not assigned to the Diagnose code.
- The Diagnose code handler entry point was not loaded.
- The wrong entry point name was specified on DEFINE DIAGNOSE or MODIFY DIAGNOSE.
- The necessary calling parameters were not defined for your entry point (such as INVAR not specified when needed).

The following commands will be useful:

- QUERY DIAGNOSE *diag*
- LOCATE DGNBK *diag*
- LOCATE SYMBOL *entry_point_name*
- QUERY ICLNAME *entry_point_name*

- **Symptom: The system abends when my Diagnose code is invoked.**

Some of the causes of this symptom are:

- All symbols used by your Diagnose code are not resolved. Note: QUERY UNRESOLVED will show any unresolved symbols that were loaded using CPXLOAD but will not show unresolved symbols that resulted during a build of the system.
- All address constants that your code uses were not resolved.
- The EPNAME value specified on the DEFINE DIAGNOSE or the MODIFY DIAGNOSE is an external symbol (for example, the CSECT name of your routine) rather than the entry point name which is coded to set up your routine's addressability.
- Your routine is not compiled with the correct level of the maclibs.

The following commands will be useful:

- LOCATE DGNBK *diag*
- QUERY UNRESOLVED
- LOCATE SYMBOL *entry_point_name*

Additional Readings

For additional information on defining new Diagnose codes, see [Chapter 6, “Defining and Modifying Commands and Diagnose Codes,”](#) on page 43.

Message Related Problems

- **Symptom: My message repository is not being used.**

Some of the causes of this symptom are:

- The entry point containing the message repository data is not associated with the correct component and language.
- Other entry points are associated ahead of the entry point with the same component and language and those entry points define the same message numbers.
- The correct component ID was not specified on the HCPCONSL invocation.

The following commands will be useful:

- QUERY CPLANGLIST ASSOCIATED
- LOCATE SYMBOL *entry_point_name*

- **Symptom: My message is being displayed with the wrong header.**

Some of the causes of this symptom are:

- The message number is in the 7000-7999 series of messages and you expect a header. Headers are not generated for these messages.
- The HCPCONSL macro was coded with the wrong operands.

The following commands will be useful:

- QUERY CPLANGLIST ASSOCIATED

Additional Readings

For additional information on adding message repositories, see [Appendix H, “Understanding the CP Message Repository,”](#) on page 209.

Exit Point Related Problems

- **Symptom: An entry point associated with an exit is not being invoked.**

Some of the causes of this symptom are:

- The exit is not enabled.
- An ASSOCIATE EXIT was not performed for the entry point.
- The wrong exit number was specified on ASSOCIATE EXIT or ENABLE EXIT. For example, you meant 1243 but you typed 1234.
- The entry point code is not loaded.
- More than one entry point is associated with an exit and a previous routine returned a control return code indicating the calling sequence should be cancelled.
- The exit point is an initialization exit (e.g. 00E0, 00E1) and the code was not CPXLOADed in the configuration file with the NODELAY option for CPXLOAD, or the code was not found on the Parmdisk.

The following commands will be useful:

- LOCATE SYMBOL *entry_point_name*
- QUERY CPXLOAD ALL
- QUERY EXIT *nnnn*

- **Symptom: The system abends when my exit is invoked.**

Some of the causes of this symptom are:

- All address constants in your code were not resolved.
- The necessary calling parameters were not defined for your entry point (e.g. MP specified instead of NONMP).
- An EPNAME value specified on ASSOCIATE EXIT is an external symbol (for example, the CSECT name of your routine) rather than an entry point name which is coded to set up your routine's addressability.
- Your routine is not compiled with the correct level of the maclibs.
- You have a lock contention problem.

The following commands will be useful:

- QUERY CPXLOAD ID *load_id*
- QUERY EXITS *exit_number*
- QUERY UNRESOLVED
- LOCATE XITBK *exit_number*

Additional Readings

For additional information on adding exit points, see [Chapter 3, “Creating a Dynamically Loaded Routine,”](#) on page 11, [Chapter 4, “Loading Dynamically into the System Execution Space,”](#) on page 33, [Chapter 5, “Controlling a Dynamically Loaded Routine,”](#) on page 39, and [Appendix J, “Samples of Dynamically Loaded Routines,”](#) on page 217.

CPXLOAD Related Problems

- **Symptom: CPXLOAD command will not load a file.**

Some of the causes of this symptom are:

- You are attempting to reload an entry point that was linked into the system at system generation time.
- You are attempting to load an entry point that is already loaded.
- The file is not on a CP-accessed disk.

The following commands will be useful:

- QUERY CPXLOAD ALL
- QUERY CPDISKS
- CPLISTFILE

- **Symptom: CPXLOAD statement will not load a file.**

Some of the causes of this symptom are:

- You are attempting to reload an entry point that was linked into the system at system generation time.
- You are attempting to load an entry point that is already loaded.
- The file is not on a CP-accessed disk.
- CPXLOAD with the NODELAY option was not specified and the code is needed by an initialization exit point.

- CPXLOAD with the NODELAY option was specified and the file is not on the Parmdisk.

The following commands will be useful:

- QUERY CPXLOAD ALL
- CPTYPE *config_file*
- QUERY CPDISKS
- CPLISTFILE

Additional Readings

For additional information on adding CPXLOAD, see [Chapter 4, “Loading Dynamically into the System Execution Space,”](#) on page 33.

CPUNXLOAD Related Problems

- **Symptom: CPXUNLOAD command will not unload a file.**

Some of the causes of this symptom are:

- You specified the wrong CPXLOAD ID.
- The files to be unloaded were loaded as PERM.
- The entry point is in use by a system service (e.g. entry point is a message repository).

The following commands will be useful:

- QUERY CPXLOAD ID *id_number*
- QUERY CPLANGLIST ASSOCIATED

Additional Readings

For additional information on adding CPXUNLOAD, see [Chapter 8, “Unloading Dynamically from the System Execution Space,”](#) on page 61, and [Chapter 5, “Controlling a Dynamically Loaded Routine,”](#) on page 39.

Miscellaneous CP Customization Errors

- **Symptom: Installation modifications converted to be CPXLOADed no longer work**

Some of the causes of this symptom are:

- All address constants in your code were not resolved.
- The necessary calling parameters were not defined for your entry point (e.g. MP specified instead of NONMP).
- Address constants in the resident nucleus attempted to reference symbols in the CPXLOADed modules.

The following commands will be useful:

- QUERY CPXLOAD ID *id_number*
- LOCATE SYMBOL *entry_point_name*
- QUERY ICLNAME *entry_point_name*

- **Symptom: "LOCATE entry_point" does not show information about my entry point.**

Some of the causes of this symptom are:

- You did not specify "LOCATE SYMBOL *entry_point_name*". If you do not specify SYMBOL, the command may be mistaken for a different locate function such as LOCATE *rdev*.

Appendix A. IBM-Defined CP Exit Points

This appendix describes each of the CP exit points defined by IBM. Valid CP exit numbers are between X'0000' and X'FFFF' with the following assignments:

CP Exit Points	Are Reserved for:
0000 - 7FFF	IBM-defined CP exit routines
8000 - EFFF	Vendors, including general distribution (for example, among SHARE customers)
F000 - FFFF	Customers for local use (for example, customers that only keep their CP exits in their own shops)

The CP exit numbers reserved for IBM (X'0000' through X'7FFF') are organized into the following groups:

CP Exit Points	Task
Initialization Processing (X'0000' through X'0FFF')	
0000 - 00FF	System initialization
0100 - 01FF	Device initialization
0200 - 02FF	Spool initialization
0300 - 0FEF	Other types of initialization
0FF0 - 0FFF	Command processing initialization
CP Command Processing (X'1000' through X'3FFF')	
1000 - 107F	AUTOLOG command
1080 - 10FF	XAUTOLOG command
1100 - 117F	LOGON command
1180 - 11FF	LOGOFF command
1200 - 120F	DIAL command
1210 - 122F	Messaging commands (MESSAGE, WARNING, SMSG)
1230 - 3FDF	Other types of CP commands
3FE0 - 3FFF	SHUTDOWN command
Resource Control Processing (X'4000' through X'6FFF')	
4000 - 41FF	Spool space
4200 - 43FF	Temporary disk space
4400 - 441F	Real spool devices
4420 - 4FFF	Other types of resource control processing
5000 - 51FF	Scheduler processing
5200 - 6FFF	Other types of resource control processing
Termination Processing (X'7000' through X'7FFF')	

Note: The most current list of CP exit point definitions and groupings is maintained in the file HCPEQXIT COPY.

Summary of IBM-Defined CP Exit Points

The following IBM-defined CP exit points are available for use on your z/VM system:

Table 5. Summary of IBM-Defined CP Exit Points

CP Exit	Name	Function	Page
00E0	Startup Pre-Prompt Processing	Provide system initialization options to CP, as if they were provided by the system operator. This CP exit point is called when CP prompts the system operator for system initialization options.	“CP Exit 00E0: Startup Pre-Prompt Processing” on page 97
00E1	Startup Post-Prompt Processing	Examine and, optionally, change the system initialization options, whether supplied by the system operator or by CP Exit 00E0, before CP validates them. This CP exit point is called after CP Exit 00E0 has been, or would have been, called. That is, after CP prompts the system operator for system initialization options and before CP validates those options.	“CP Exit 00E1: Startup Post-Prompt Processing” on page 101
OFFB	Post-Authorization Command Processing	Examine and, optionally, change or reject any CP command that has already passed system authorization processing. This CP exit point is called after CP has performed authorization processing for a CP command (or subcommand) but before CP has parsed any of the command operands.	“CP Exit OFFB: Post-Authorization Command Processing” on page 105
1100	LOGON Command Pre-Parse Processing	<p>Examine and, optionally, change or reject the CP LOGON command before CP parses the command. This CP exit point is called immediately after a user enters the CP LOGON command, but before CP parses the command. You can use this CP exit point to:</p> <ul style="list-style-type: none">• Specify CP LOGON command operands to:<ul style="list-style-type: none">– Be added to the list of user-specified LOGON operands– Replace the list of user-specified LOGON operands• Replace the specified console input data• Reject the LOGON request.	“CP Exit 1100: LOGON Command Pre-Parse Processing” on page 109
1101	Logon Post-Parse Processing	<p>Examine and, optionally, change the return code generated by CP after parsing the CP LOGON command. This CP exit point is called after CP parses a CP LOGON command, but before CP examines the return code from the parse (which CP uses to direct the rest of the logon process). You can use this CP exit point to:</p> <ul style="list-style-type: none">• Examine the return code from the parse• Change the return code from the parse.	“CP Exit 1101: Logon Post-Parse Processing” on page 112

Table 5. Summary of IBM-Defined CP Exit Points (continued)

CP Exit	Name	Function	Page
1110	VMDBK Pre-Logon Processing	Examine and, optionally, change the virtual machine definition block (VMDBK) for a virtual machine after CP creates and initializes it. This CP exit point is called during virtual machine creation after CP has: <ul style="list-style-type: none"> • Created and initialized the VMDBK, • Defined all the virtual CPUs, • Established all the virtual devices, and • Created the base address space. 	“CP Exit 1110: VMDBK Pre-Logon Processing” on page 114
117F	Logon Final Screening	Examine and, optionally, perform any processing required immediately before a successful LOGON command completes. Perform any processing required immediately before a successful LOGON command completes. This CP exit point is called after CP completes all logon processing, but immediately before terminating the execution of the LOGON command.	“CP Exit 117F: Logon Final Screening” on page 115
11C0	Logoff Initial Screening	Examine the CP LOGOFF command immediately after it is issued. This CP exit point is called immediately after a CP LOGOFF command is issued and CP has set the appropriate status flags in the VMDBK, but before CP has: <ul style="list-style-type: none"> • Displayed the LOGOFF message, • Terminated console spooling, • Performed any other cleanup processing associated with logging off a virtual machine. 	“CP Exit 11C0: Logoff Initial Screening” on page 117
11FF	Logoff Final Screening	Examine and, optionally, perform any processing required immediately before a successful LOGOFF command completes. This CP exit point is called after CP completes most logoff processing, but before CP terminates the execution of the LOGOFF command. At this point, the virtual machine definition block (VMDBK) still exists, but CP has released most of the data areas to which it referred.	“CP Exit 11FF: Logoff Final Screening” on page 118
1200	DIAL Command Initial Screening	Examine and, optionally, change or reject the operands specified on the CP DIAL command. This CP exit point is called immediately after a user enters the command and before CP processes it.	“CP Exit 1200: DIAL Command Initial Screening” on page 120
1201	DIAL Command Final Screening	Examine and, optionally, reject the decisions made by the processing of the CP DIAL command. This CP exit point is called immediately after CP has processed the DIAL command operands and knows the target user ID and the target virtual device number, but before CP completes the actual connection. Using this CP exit point, you can only examine the decisions and their values and choose whether to accept this DIAL command or to reject it. You cannot use this CP exit point to change any of the decisions or their values.	“CP Exit 1201: DIAL Command Final Screening” on page 122

Table 5. Summary of IBM-Defined CP Exit Points (continued)

CP Exit	Name	Function	Page
1210	Messaging Commands Screening	Examine and, optionally, change or reject one of the following messaging commands: MESSAGE, MSGNOH, SMSG, and WARNING.	“CP Exit 1210: Messaging Commands Screening” on page 123
1230	VMRELOCATE Eligibility Checks	Define installation-specific conditions for disallowing the relocation of virtual machines using the VMRELOCATE command in a single system image.	“CP Exit 1230: VMRELOCATE Eligibility Checks” on page 129
1231	VMRELOCATE Destination Restart	Use CP Exit 1231 to do any necessary processing before the target guest is restarted on the destination system during a live guest relocation.	“CP Exit 1231: VMRELOCATE Destination Restart” on page 130
3FE8	SHUTDOWN Command Screening	Examine and, optionally, change the operands of the CP SHUTDOWN command. This CP exit point is called immediately before CP begins its examination of the operands specified on the SHUTDOWN command.	“CP Exit 3FE8: SHUTDOWN Command Screening” on page 131
4400	Separator Page Data Customization	Examine and, optionally, modify the information that will be printed on the VM separator page for printed output. This CP exit point is called during the processing of VM separator pages, after CP fills in the information that will be printed on the separator page, but before CP prints it.	“CP Exit 4400: Separator Page Data Customization” on page 133
4401	Separator Page Pre-Perforation Positioning	Change the channel program which positions impact printers at the bottom of a form for printing across the perforation.	“CP Exit 4401: Separator Page Pre-Perforation Positioning” on page 135
4402	Separator Page Perforation Printing or 3800 Positioning	Change the channel program which either: <ul style="list-style-type: none"> Prints lines of asterisks and dashes across the perforation of separator pages for impact printers, and positions the printer for printing of separator page data Positions 3800 printers for printing of separator page data 	“CP Exit 4402: Separator Page Perforation Printing or 3800 Positioning” on page 137
4403	Separator Page Printing	Change the channel program which controls the printing of separator page data.	“CP Exit 4403: Separator Page Printing” on page 138
4404	Second Separator Page Positioning	Change the channel program which positions the printer for printing of the second separator page. This channel program either positions impact printers at the bottom of the first separator page to repeat perforation printing, or positions 3800 printers to the top of the second separator page.	“CP Exit 4404: Second Separator Page Positioning” on page 140
4405	Second Separator Page Printing	Change the channel program which controls printing of the second separator page.	“CP Exit 4405: Second Separator Page Printing” on page 142
4406	Separator Page Post-Print Positioning	Change the channel program which positions the printer to begin printing spool file data after the separator pages have been printed.	“CP Exit 4406: Separator Page Post-Print Positioning” on page 143

Table 5. Summary of IBM-Defined CP Exit Points (continued)

CP Exit	Name	Function	Page
4407	Trailer Page Processing	Change or replace data which is printed on the separator trailer page, and/or to modify the channel program which controls printing of the separator trailer page.	“CP Exit 4407: Trailer Page Processing” on page 145

Each of these CP exit points is described in more detail in the following pages. These descriptions include the following information, where applicable:

- Usage conventions
- Standard point of processing
- Standard entry conditions
- Standard parameter list contents
- Standard exit conditions
- Standard return codes.

Usage Conventions

This section describes the standard conventions used to pass control to and from the IBM-defined CP exit points. Your dynamically loaded CP exit routines should follow these conventions.

Standard Point of Processing

This section describes:

- Basic information about:
 - How the CP exit is used (reentrant or serially reusable)
 - Type of processing (MP or non-MP)
 - Locks (does the calling module have any locks in place).
- The events that led up to this CP exit point being called. (For example, someone issues a command which causes a CP exit to be called to process the command.)
- The conditions and actions upon return. (For example, if CP receives the proper return codes upon return, it would process the CP exit task and return control to the calling task.)

Standard Entry Conditions

When CP passes control to each CP exit routine, the registers contain the following information:

Register	Contents
R0	CP exit number
R1	Address of the parameter list
R2	Address of the CP exit call request block (XCRBK)
R3-R10	Undefined
R11	If called during initialization, this register is undefined. If called after initialization, the contents of this register will depend on the specific CP exit point, but will frequently be the address of the virtual machine definition block (VMDBK).
R12	Module base register
R13	Address of a dynamic SAVBK (call with savearea block).

Register Contents

R14-R15 Undefined

Standard Parameter List Contents

Most CP exit points place the address of the parameter list in R1. This section lists the information that is usually included in a parameter list. Refer to the specific CP exit point documentation for information on the particular items provided in that exit's parameter list. HCPPLXIT COPY contains DSECTS for parameter list definitions for IBM-defined CP exit points.

The standard parameter list contains:

Xxxxx	DSECT	,	
	DS	A	Reserved
XxxxxUWD	DS	A	Address of 32 bytes of user ... words doubleword aligned ... initialized to binary ... zeros
XxxxxTRT	DS	A	Address of 256 bytes ... words doubleword aligned ... initialized to binary ... zeros
Xxxxx...	DS	A	Address of a data item
Xxxxx...	DS	A	Address of a data item

For exit points defined by the HCPXSERV macro with the PLIST operand, the HCPXSERV macro turns on the high order bit of the last address in the parameter list. For maximum safety, an exit routine should use an address in the parameter list only if no prior address is marked as the last entry.

Storage in the parameter list is initialized by the HCPXSERV macro. The parameter list and the storage to which it points are untouched by CP code that processes the calling of exit routines. If more than one exit routine is associated to an exit point, each exit routine sees whatever the prior exit routine modified.

Standard Exit Conditions

When most CP exit routines return control to CP, the registers contain the following information:

Register Contents

R0-R14 Restored to their original contents

R15 Always contains a return code.

Standard Return Codes

Return codes tell CP how it should continue to process the task from which the CP exit routine was called. Each CP exit routine issues 2 return codes, each of which must be a multiple of 4, in R15 when it returns control to CP. The return code in bytes 0 through 1 is called the "exit control return code". The return code in bytes 2 through 3 is called the "mainline return code".

Exit Control**Return Code Results**

0	CP calls the next entry point associated with this CP exit point and continues the task.
4	CP skips the next entry point associated with this CP exit point. CP calls the following entry point associated with this CP exit point and continues the task.
8	CP discontinues calling any remaining entry points associated with this CP exit point. CP returns to the mainline routine, passing back the maximum "mainline return code" return by the CP exit routines that were called.

Exit Control Return Code	Results
12	CP discontinues calling any remaining entry points associated with this CP exit point. CP returns to the mainline routine, passing back the "mainline return code" returned by the last CP exit routine that was called.
<i>nn</i>	If any other "exit control return code" is returned to CP, CP: <ol style="list-style-type: none"> 1. Generates soft abend ZXU001, 2. Writes message HCP2765E to the operator, and 3. Continues as if "exit control return code" 8 had been returned.

After calling all entry points associated with this CP exit point, CP returns to the mainline routine, passing back the maximum "mainline return code" returned by the CP exit routines that were called.

The processing of "mainline return codes" varies for each CP exit point. However, return code 0 typically tells CP to proceed as if no CP exit routine had been called. Other "mainline return codes" typically direct CP to change its normal processing. See the description of each CP exit point for more information about "mainline return codes".

For dynamic exits, the "mainline return code" determines whether or not the instruction at the exit point is executed. Return code 0 tells CP to execute the instruction; any other value tells CP to skip the instruction.

Additional Information about Control Entry Points

Control entry points are called during CPXLOAD and CPXUNLOAD command processing when the CONTROL operand is specified. They are also called during system configuration file processing when a CPXLOAD statement specifies the CONTROL operand. The environment in which they execute is different from that of dynamically loaded CP exit routines.

Entry Conditions

When CP passes control to the control entry point, the registers contain the following information:

Register	Contents
R0	Undefined
R1	Address of HCPLABK. This control block contains lots of information about the CPXLOAD or CPXUNLOAD request.
R2	1 Control entry point was called as a result of a CPXLOAD request 2 Control entry point was called as result of CPXUNLOAD request.
R3-R10	Undefined
R11	See table below
R12	Entry point address
R13	Address of SAVBK
R14-R15	Undefined

The control entry point execution environment varies depending on how the control entry point is called. The following table describes these different execution environments.

Situation when control entry point is called	Execution Environment
CPXLOAD statement in system configuration file that specifies the NODELAY operand	<p>The routine is executed very early during CP initialization. Many CP services are not yet initialized. For instance, there is no real console on which to display messages. If you wish to display a message, you should code the HCPCONSL macro to generate the message in an GSDBK and add that GSDBK to the queue anchored by HCPISUSC. Messages on this queue are display during CP initialization after the IBM copyright banner.</p> <p>R11 contains the SYSTEM VMDBK address. Do not use system services that rely on a logged-on user ID.</p> <p>Dynamically loaded CP routines must be in a file on the parm disk, which is the only disk that CP has access to at this stage of the initialization process.</p>
CPXLOAD statement in system configuration file with the DELAY operand (or defaulted)	<p>The routine is executed during CP initialization. The directory and CP_ACCESS statements in the system configuration file have been processed. Many CP services are not yet initialized. The HCPCONSL macro can be used to display messages on the IPL console.</p> <p>R11 contains the address of a skeleton VMDBK for the OPERATOR.</p> <p>Dynamically loaded CP routines must be located on one of the disks specified on a CP_ACCESS statement.</p>
CPXLOAD command	<p>CP has finished initialization. All CP services are available.</p> <p>R11 contains the issuing user's VMDBK address. CP services relying on a logged-on user ID (for example, HCPCONSL) will work.</p>
CPXUNLOAD command with the ASYNC operand specified (or defaulted)	<p>CP has finished initialization. All CP services are available.</p> <p>R11 contains the SYSTEM VMDBK address. Do not use system services that rely on a logged-on user ID. HCPCONSL requests should specify the "DESTINATION=" operand.</p>
CPXUNLOAD command with the SYNC operand specified	<p>CP has finished initialization. All CP services are available.</p> <p>R11 contains the issuing user's VMDBK address. CP services relying on a logged-on user ID (for example, HCPCONSL) will work.</p>

Other Entry Conditions

During CPXLOAD processing:

- If the control entry point that is specified cannot be found, the CPXLOAD request is rejected.
- If the control entry point is in a TEMPORARY routine that is not part of the current CPXLOAD, CP rejects the CPXLOAD request. This is because the control entry point might later be unloaded and CP would not be able to call it during CPXUNLOAD processing.
- The DLUCPXLK is held exclusively. The control entry point must not relinquish the lock or CP may find another CPXLOAD or CPXUNLOAD that is waiting to run, and its view of what external symbols exist may become out of date.
- All address constants that can be resolved have been resolved. This means that address constants inside the routines that CP just loaded have been resolved if they refer to anything that CP just loaded or

if they refer to anything that is considered PERMANENT. If they refer to something that CP did not just load and that is considered TEMPORARY, then the CPXLOAD request would have failed.

- No external symbols loaded as part of this CPXLOAD operation have been added to the CP symbol map. If the control entry point needs to call an entry point that is part of what this CPXLOAD operation is loading, it must use a direct call specifying the address of the entry point. The address can be obtained by using a simple address constant or by calling HCPCYEES to search for the ESD entry and using the address from ESDBVADR in the found HCPESDBK.

During CPXUNLOAD processing:

- The DLUCPXLK is held exclusively. The control entry point must not relinquish the lock or CP may find another CPXLOAD or CPXUNLOAD that is waiting to run, and its view of what external symbols exist may become out of date.
- The control entry point can use indirect calls to other entry points.
- The control entry point is called after CP has already checked whether the code can be unloaded (TEMPORARY, not in use). If the control entry point signals that the unload should continue, CP will unload the code.

Exit Conditions

When the control entry point returns control to CP, the routine must return a return code in SAVER15. The valid return codes are:

Return Code	Meaning
0	CP should continue the CPXLOAD or CPXUNLOAD request.
4	CP should reject the CPXLOAD or CPXUNLOAD request.

Debugging Your Control Entry Point Routine

Because a control entry point is loaded and run before you have a chance to use the CP LOCATE command to find it in storage, it will not be as easy to trace and debug your routine. However, there are several tracing and debugging techniques that you can use to debug your control entry point routine on a second-level system:

1. To trace a control entry point routine during CPXLOAD, set a trace point in HCPCLG where it calls the control entry point routine. To trace the routine during CPXUNLOAD, set a trace point in HCPCLH where it calls the control entry point routine.
2. At the beginning of your routine, insert an instruction that you know is not issued anywhere else in CP: for example, an SVC instruction that CP does not support (such as SVC 1) or an EXECUTE of an EXECUTE instruction (EX R0,*). Establish a breakpoint for that instruction or program check (TRACE SVC 1, or TRACE PROG 3). When the breakpoint is reached, establish any other desired TRACE entries. Then resume execution after the instruction that triggered the original breakpoint (BEGIN aaaaaa).

Be warned that instructions inserted by this technique cannot be part of a first-level system. The interrupt cannot be trapped, and CP will crash. Also remember to delete such instructions from the final version of your routine.

CP Exit 00E0: Startup Pre-Prompt Processing

Purpose

Use CP Exit 00E0 to provide system initialization options to CP, as if they were provided by the system operator.

Point of processing

Process	CP Exit Attribute
Startup Pre-Prompt Processing	<ul style="list-style-type: none">• Reentrant• MP• No locks held

CP Exit 00E0 is called whenever CP prompts the system operator for system initialization options. (For an explanation of when prompting normally occurs, see Note “1” on page 99 in the Programming Considerations section.) The calling task provides data areas where the CP exit routines can store the CP exit-supplied system initialization options.

Entry conditions

Register 1 points to a parameter list which is described below. For more information about other register contents, see “Standard Entry Conditions” on page 93.

Parameter list constant

The parameter list contains the following:

Word	Offset	Data Length	Contents
1	+0	N/A	Reserved for future IBM use.
2	+4	32 bytes	Address of the doubleword-aligned User Area, which can be used by this CP exit routine as working storage.
3	+8	256 bytes	Address of the doubleword-aligned translate-and-test (TRT) table, which can be used by this CP exit routine as working storage.
4	+12	4 bytes	Address of the fullword-aligned maximum length of the system initialization options area (Word 6).
5	+16	4 bytes	Address of the fullword-aligned actual length of the system initialization options area (Word 6), which this CP exit routine uses as a place to store the actual length of the information in the area located by Word 6. If the CP exit routines places any data into the area located by Word 6, then the CP exit routine must store the length of the data into this length field (Word 5).
6	+20	(See Word 5)	Address of the system initialization options area, which can be used by this CP exit routine as a place to store the options CP should use during system initialization.
7	+24	4 bytes	Address of the fullword-aligned general system data block (GSDBK) that can be allocated and used by this CP exit routine in place of the system initialization options area (Word 6). If the CP exit routine creates a GSDBK to contain the system initialization options, then the CP exit routine must place the address of the GSDBK into this field.

Exit conditions

On return, CP Exit 00E0 sets the standard register contents described in “Standard Exit Conditions” on page 94.

Return codes

On return, R15 contains two return codes. The first half of the fullword contains the "CP exit control return code", which is explained in "Standard Return Codes" on page 94. The second half of the fullword contains the "mainline return code", which is explained in the following table:

Return Code	Results
0	CP ignores any system initialization options supplied by this CP exit routine, prompts the system operator for system initialization options, and continues normal initialization processing.
4	CP converts the system initialization options in the data area located by Word 6 into a GSDBK, parses the options as if the system operator had entered the data, and continues normal initialization processing. CP does not prompt the system operator for the system initialization options.
8	CP uses the system initialization options in the GSDBK (located by Word 7), parses the options as if the system operator had entered the data, and continues normal initialization processing. CP does not prompt the system operator for the system initialization options.
<i>nn</i>	For any other return codes, CP: <ul style="list-style-type: none">• Writes message HCP2765E to the operator• Disables CP Exit 00E0• Prompts the system operator for the system initialization options.

Programming considerations

1. If enabled, CP Exit 00E0 is called whenever CP would normally prompt the system operator for system initialization options. This normally occurs in the following situations:

- During a system IPL when the system configuration file does not enable the automatic warm start function using the following statement:

```
Features Enable Auto_Warm_IPL
```

- During a system IPL when the system operator specifies the IPL parameter PROMPT, which overrides the automatic warm start function
- During a system restart, after a CP abend, if the system operator tells CP to perform system initialization prompting using one of the following:
 - System configuration file statement:

```
Features Enable Prompt After_Restart
```

- CP command:

```
set prompt after_restart on
```

- During an operator-initiated system re-IPL if the system operator tells CP to perform system initialization prompting using one of the following:

- System configuration file statement:

```
Features Enable Prompt After_Shutdown_ReIPL
```

- CP command:

```
set prompt after_shutdown_reipl on
```

2. CP Exit 00E0 is called early in the system initialization process. The only minidisk that is available to CP at that time is the parmdisk. This means your exit routines for CP Exit 00E0 must be placed on the parmdisk. In addition, you must load and enable this CP exit point in the system configuration file. To do so, use the:
 - a. CPXLOAD statement with the NODELAY option to dynamically load the CP routines on the parmdisk into the system execution space
 - b. ASSOCIATE EXIT statement to assign one or more entry points or external symbols to this CP exit point
 - c. ENABLE EXITS statement to enable this CP exit point.

You cannot use the CP commands of the same name to load and enable CP Exit 00E0, because system initialization will have completed by the time you can log on and issue the commands.

3. **Availability Status of CP Services** — Because CP Exit 00E0 is called early in the system initialization process, some CP Services will not be available:
 - The user directory will not yet have been read.
 - The spooling subsystem will not yet have been initialized.
 - You can only call routines in modules that:
 - You loaded using the CPXLOAD statement and the NODELAY option
 - Are located in the CP nucleus.
 - During most of system initialization, CP will be running in uniprocessor mode. Any alternate processors will not yet have been brought online.
4. After CP Exit 00E0 completes, CP uses the mainline return code to decide what to do next. For return code 0, CP prompts the system operator for the system initialization options. For return codes 4 and 8, CP stores the system initialization options that CP Exit 00E0 provided.

Then, CP tries to call CP Exit 00E1 (if it is enabled) to validate the system initialization options or to provide additional options. After CP Exit 00E1 completes, CP parses all the system initialization options. If CP finds anything wrong with the options, CP repeats the entire process for gathering system initialization information.

If your CP Exit 00E0 routine provided CP with the system initialization options that were not valid, your z/VM system will be caught in a loop and will not initialize. To prevent this looping condition, we suggest you code your CP exit routine to:

- Examine the XCRCALLS field in the exit call request block (XCRBK). (The address of the XCRBK is passed in Register 2.) The XCRCALLS field counts how many times a CP exit routine has been called by and returned to CP. If the value in XCRCALLS is greater than 0 (zero), then this CP exit routine has been called before and is being called again.

To break the loop, consider adding code to your CP exit routine that tests to see if the XCRCALLS field is greater than zero. If the answer is yes, have your CP exit routine set the mainline return code to 0, thereby forcing CP to ignore the system initialization options supplied by CP Exit 00E0 and prompt the system operator for the options. This XCRCALLS check will allow the system operator to correct the system initialization options.

If your CP Exit 00E1 routine provided CP with the system initialization options that were not valid, you should consider including code in that routine to prevent the looping condition.

- Disable itself before returning to CP. You are only prompted for system initialization options at one point in the system initialization process. If CP Exit 00E0 is called more than once, it is because one or more of the system initialization options were not valid.

To prevent the loop, consider coding the HCPXSERV DISABLE macro as the very last entry in your CP exit routine. Disabling this CP exit routine just before you return to CP will not have any effect on:

- The system initialization options that you have already stored in the data area or GSDBK
- Your running z/VM system, because CP Exit 00E0 is not called after system initialization

- The next IPL, because you are loading and enabling CP Exit 00E0 in the system configuration file.

For more information on the HCPXSERV macro, see page [“HCPXSERV: CP Exit Services Director”](#) on page 170.

- 5. Tracing and Debugging** — Because CP Exit 00E0 is called early in the system initialization process, it will not be as easy to trace and debug this CP exit routine as it would be for a CP exit routine that is called after initialization. During system initialization, normal CP commands (such as LOCATE and LOCK) are not yet available to you. However, there are a number of tracing and debugging techniques that you can use to locate your CP exit code on a second-level system:

Method 1 —

At entry point HCPISXE0 in module HCPISX, set a trace point. At this point in HCPISX processing, all the CPXLOAD statements that specified the NODELAY operand will have been processed and all your modules will have been loaded.

During initialization when CP executes the trace point that you set at entry point HCPISXE0, your system drops into CP READ status. In CP READ, you can issue any CP command (for example, LOCATEVM or TRACE). Use the CP LOCATEVM command to find where your CP exit routine was loaded. For example, if your CP exit routine used a label of EXITE0 on the HCPPROLG macro, you could find it using this command:

```
locatevm 4-end case ignore increment 1000 data exite0
```

Method 2 —

At the beginning of your module, insert instructions to change and then reset register 11. R11 generally contains the address of the dispatched virtual machine definition block (VMDBK). As a rule, you should never change R11 in a CP exit routine. However, if changed judiciously, you can change R11 safely. For example, if you added the following instructions at the beginning of your module, you could set a TRACE breakpoint for the alteration of R11 to a predictable value:

LR	R4,R11	Save current value of R11
LR	R11,R0	Alter R11 to the CP exit number
LR	R11,R4	Restore R11 to its original value

Before you IPL the system, establish a breakpoint for the modification of R11 to an expected value:

```
CP TRACE GB DATA 000000E0 <--- Stop when R11 equals our CP exit number
IPL nnn CLEAR          <--- the address of the IPL DASD
```

In this case, the changed value for R11 would be the CP exit number in R0. Be warned, while this technique is workable, it can be painfully slow.

Method 3 —

At the beginning of your module, insert an instruction known not to be issued anywhere else in CP. For example, an SVC instruction that CP does not support (SVC 1) or an EXECUTE of an EXECUTE instruction (EX R0,*). Establish a breakpoint for that instruction or program check (TRACE SVC 1, or TRACE PROG 3). When the breakpoint is reached, establish any other desired TRACE entries. Then, resume execution after the instruction that triggered the original breakpoint (BEGIN zzzzzz).

Be warned that instructions inserted by this technique cannot be part of a first-level system. The interrupt cannot be trapped, and CP will crash.

Once you find your CP exit routine, you can use the CP TRACE command to trace its execution.

CP Exit 00E1: Startup Post-Prompt Processing

Purpose

Use CP Exit 00E1 to examine and, optionally, change the system initialization options, whether supplied by the system operator or by CP Exit 00E0, before CP validates them. This allows your CP exit routine to:

- Supply additional CP-defined system initialization options, such as DRAIN or DISABLE
- Supply installation-defined system initialization options
- Perform additional authorization checking before allowing a clean or cold start.

Point of processing

Process	CP Exit Attribute
Startup Post-Prompt Processing	<ul style="list-style-type: none"> • Reentrant • MP • No locks held

CP Exit 00E1 is called after CP Exit 00E0 has been, or would have been, called. That is, after CP prompts the system operator for system initialization options and before CP validates those options. (For an explanation of when prompting normally occurs, see Note “1” on page 103 in the Programming Considerations section.) The calling task provides the system initialization options that were supplied to CP, whether by the system operator or by CP Exit 00E0.

Entry conditions

Register 1 points to a parameter list which is described below. For more information about other register contents, see “Standard Entry Conditions” on page 93.

Parameter list constant

The parameter list contains the following:

Word	Offset	Data Length	Contents
1	+0	N/A	Reserved for future IBM use.
2	+4	32 bytes	Address of the doubleword-aligned User Area, which can be used by this CP exit routine as working storage.
3	+8	256 bytes	Address of the doubleword-aligned translate-and-test (TRT) table, which can be used by this CP exit routine as working storage.
4	+12	4 bytes	Address of the fullword-aligned maximum length of the system initialization options area (Word 6).
5	+16	4 bytes	Address of the fullword-aligned actual length of the system initialization options area (Word 6), CP Exit 00E1 can change this length value.
6	+20	(See Word 5)	Address of the system initialization options area.
7	+24	4 bytes	Address of the fullword-aligned general system data block (GSDBK), which can be allocated and used by this CP exit routine in place of the system initialization options area (Words 6). If the CP exit routine creates a GSDBK to contain the system initialization options, then the CP exit routine must place the address of the GSDBK into this field.

Exit conditions

On return, CP Exit 00E1 sets the standard register contents described in “Standard Exit Conditions” on page 94.

Return codes

On return, R15 contains two return codes. The first half of the fullword contains the "CP exit control return code", which is explained in "Standard Return Codes" on page 94. The second half of the fullword contains the "mainline return code", which is explained in the following table:

Return Code	Results
0	CP ignores any system initialization options supplied by this CP exit routine and continues normal processing.
4	CP converts the system initialization options in the data area located by Word 6 into a GSDBK, parses the options as if the system operator had entered the data, and continues normal initialization processing.
8	CP uses the system initialization options in the GSDBK (located by Word 7), parses the options as if the system operator had entered the data, and continues normal initialization processing.
12	<p>CP ignores any system initialization options supplied by this CP exit routine, reprompts the system operator for system initialization options, and continues normal initialization processing.</p> <p>Note: Return code 12 indicates that there is something wrong with the system initialization options. If you have CP Exit 00E0 enabled and supplying the system initialization options, you should be very careful that you do not code a situation that results in an infinite loop.</p>
<i>nn</i>	<p>For any other return codes, CP:</p> <ul style="list-style-type: none">• Writes message HCP2765E to the operator• Disables CP Exit 00E1• Uses the system initialization options stored in the area located by Word 6, ignoring any changes made by this CP exit routine.

Programming considerations

1. If enabled, CP Exit 00E1 will be called before CP would normally begin to examine the system initialization options. This normally occurs in the following situations:

- During a system IPL when the system configuration file does not enable the automatic warm start function using the following statement:

```
Features  Enable  Auto_Warm_IPL
```

- During a system IPL when the system operator specifies the IPL parameter PROMPT, which overrides the automatic warm start function
- During a system restart, after a CP abend, if the system operator tells CP to perform system initialization prompting using one of the following:

- System configuration file statement:

```
Features  Enable  Prompt  After_Restart
```

- CP command:

```
set prompt after_restart on
```

- During an operator-initiated system re-IPL if the system operator tells CP to perform system initialization prompting using one of the following:
 - System configuration file statement:

```
Features  Enable  Prompt  After_Shutdown_ReIPL
```

– CP command:

```
set prompt after_shutdown_reipl on
```

2. CP Exit 00E1 is called early in the system initialization process. The only minidisk that is available to CP at that time is the parmdisk. This means your exit routines for CP Exit 00E1 must be placed on the parmdisk. In addition, you must load and enable this CP exit point in the system configuration file. To do so, use the:

- a. CPXLOAD statement with the NODELAY option to dynamically load the CP routines from the parmdisk into the system execution space
- b. ASSOCIATE EXIT statement to assign one or more entry points or external symbols to this CP exit point
- c. ENABLE EXITS statement to enable this CP exit point.

You cannot use the CP commands of the same name to load and enable CP Exit 00E1, because system initialization will have completed by the time you can log on and issue the commands.

3. **Availability Status of CP Services** — Because CP Exit 00E1 is called early in the system initialization process, some CP Services will not be available:

- The user directory will not yet have been read.
- The spooling subsystem will not yet have been initialized.
- You can only call routines in modules that:
 - You loaded using the CPXLOAD statement and the NODELAY option
 - Are located in the CP nucleus.
- During most of system initialization, CP will be running in uniprocessor mode. Any alternate processors will not yet have been brought online.

4. After CP Exit 00E1 completes, CP parses all the system initialization options. These options were supplied by either the system operator (in response to the startup prompt), CP Exit 00E0, or CP Exit 00E1. If CP finds anything wrong with the options, CP repeats the entire process for gathering system initialization information.

If your CP Exit 00E1 routine provided CP with the system initialization options that were not valid, your z/VM system will be caught in a loop and will not initialize. To prevent this looping condition, we suggest you code your CP exit routine to:

- Examine the XCRCALLS field in the exit call request block (XCRBK). (The address of the XCRBK is passed in Register 2.) The XCRCALLS field counts how many times a CP exit routine has been called by and returned to CP. If the value in XCRCALLS is greater than 0 (zero), then this CP exit routine has been called before and is being called again.

To break the loop, consider adding code to your CP exit routine that tests to see if the XCRCALLS field is greater than a small number (for example, 3). If the answer is yes, have your CP exit routine set the mainline return code to 0, thereby forcing CP to ignore the system initialization options supplied by CP Exit 00E1 and continue with the options previously provided by the system operator or CP Exit 00E0. This XCRCALLS check will allow the system operator to correct the system initialization options on a subsequent prompt.

- Disable itself before returning to CP. You are only prompted for system initialization options at one point in the system initialization process. If CP Exit 00E1 is called more than once, it is because one or more of the system initialization options were not valid.

To prevent the loop, consider coding the HCPXSERV DISABLE macro as the very last entry in your CP exit routine. Disabling this CP exit routine just before you return to CP will not have any effect on:

- The system initialization options that you have already stored in the data area or GSDBK
- Your running z/VM system, because CP Exit 00E1 is not called after system initialization

- The next IPL, because you are loading and enabling CP Exit 00E1 in the system configuration file.

For more information on the HCPXSERV macro, see [“HCPXSERV: CP Exit Services Director”](#) on page 170.

- 5. Tracing and Debugging** — Because CP Exit 00E1 is called early in the system initialization process, it will not be as easy to trace and debug this CP exit routine as it would be for a CP exit routine that is called after initialization. During system initialization, normal CP commands (such as LOCATE and LOCK) are not yet available to you. However, there are a number of tracing and debugging techniques that you can use to locate your CP exit code on a second-level system:

Method 1 —

At entry point HCPISXE1 in module HCPISX, set a trace point. At this point in HCPISX processing, all the CPXLOAD statements that specified the NODELAY operand will have been processed and all your modules will have been loaded.

During initialization when CP executes the trace point that you set at entry point HCPISXE1, your system drops into CP READ status. In CP READ, you can issue any CP command (for example, LOCATEVM or TRACE). Use the CP LOCATEVM command to find where your CP exit routine was loaded. For example, if your CP exit routine used a label of EXITE1 on the HCPPROLG macro, you could find it using this command:

```
locatevm 4-end case ignore increment 1000 data exite1
```

Method 2 —

At the beginning of your module, insert instructions to change and then reset register 11. R11 generally contains the address of the dispatched virtual machine definition block (VMDBK). As a rule, you should never change R11 in a CP exit routine. However, if changed judiciously, you can change R11 safely. For example, if you added the following instructions at the beginning of your module, you could set a TRACE breakpoint for the alteration of R11 to a predictable value:

LR	R4,R11	Save current value of R11
LR	R11,R0	Alter R11 to the CP exit number
LR	R11,R4	Restore R11 to its original value

Before you IPL the system, establish a breakpoint for the modification of R11 to a predictable value:

```
CP TRACE GB DATA 000000E1 <--- Stop when R11 equals our CP exit number
IPL nnn CLEAR          <--- the address of the IPL DASD
```

In this case, the changed value for R11 would be the CP exit number in R0. Be warned, while this technique is workable, it can be painfully slow.

Method 3 —

At the beginning of your module, insert an instruction known not to be issued anywhere else in CP. For example, an SVC instruction that CP does not support (SVC 1) or an EXECUTE of an EXECUTE instruction (EX R0,*). Establish a breakpoint for that instruction or program check (TRACE SVC 1, or TRACE PROG 3). When the breakpoint is reached, establish any other desired TRACE entries. Then, resume execution after the instruction that triggered the original breakpoint (BEGIN zzzzzz).

Be warned that instructions inserted by this technique cannot be part of a first-level system. The interrupt cannot be trapped, and CP will crash.

Once you find your CP exit routine, you can use the CP TRACE command to trace its execution.

CP Exit OFFB: Post-Authorization Command Processing

Purpose

Use CP Exit OFFB to examine and, optionally, change or reject any CP command that has already passed system authorization processing.

Point of processing

Process	CP Exit Attribute
Post-Authorization Command Processing	<ul style="list-style-type: none">• Reentrant• MP• No locks held

CP Exit OFFB is called after CP has performed authorization processing for a CP command (or subcommand) but before CP has parsed any of the command operands. You can use this CP exit point to examine, change, or reject the CP command. CP provides data areas where the CP exit routine can examine or change the command (or subcommand) operands.

Entry conditions

Register 1 points to a parameter list which is described below. For more information about other register contents, see [“Standard Entry Conditions”](#) on page 93.

Parameter list constant

The parameter list contains the following:

Word	Offset	Data Length	Contents
1	+0	N/A	Reserved for future IBM use.
2	+4	32 bytes	Address of the doubleword-aligned User Area, which can be used by this CP exit routine as working storage.
3	+8	256 bytes	Address of the doubleword-aligned translate-and-test (TRT) table, which can be used by this CP exit routine as working storage.
4	+12	12 bytes	Address of the CP command name, as entered by the user.
5	+16	1 byte	Address of a value that indicates the command type. The possible values are: Value Meaning X'01' or ACIXAC CP command other than QUERY or SET X'02' or ACIXAQ CP QUERY command followed by a keyword other than VIRTUAL, such as QUERY SET X'03' or ACIXAQV CP QUERY VIRTUAL command followed by a keyword, such as QUERY VIRTUAL PRT or QUERY VIRTUAL ALL X'04' or ACIXAQVM CP QUERY or CP QUERY VIRTUAL command without a following keyword, such as QUERY <i>userid</i> or QUERY VIRTUAL <i>vaddr</i> X'05' or ACIXAS CP SET command

Word	Offset	Data Length	Contents
6	+20	12 bytes	Address of the full base command name. That is, if the user enters a command, this is the full name of that command. If the user enters an alias, this is the full base command name of the CP command to which the alias points, not the full name of the alias.
7	+24	4 bytes	Address of the length of the user-supplied command operands located by Word 9, up to but not including the logical line-end character (X'15').
8	+28	4 bytes	Address of the length of the user-supplied command operands located by Word 9, through and including all logical line-end characters (X'15').
9	+32	(See Word 8)	Address of the user-supplied command operands, through and including all logical line-end characters (X'15').
10	+36	4 bytes	Address of the maximum length of the area containing the CP exit-supplied command (or subcommand) operands located by Word 13, through and including all logical line-end characters (X'15').
11	+40	4 bytes	Address of the length of the area for new CP exit-supplied command (or subcommand) operands located by Word 13, up to but not including the logical line-end character (X'15').
12	+44	4 bytes	Address of the length of the area for new CP exit-supplied command (or subcommand) operands located by Word 13, through and including all logical line-end characters (X'15').
13	+48	(See Word 12)	Address of the area for new CP exit-supplied command (or subcommand) operands, through and including all logical line-end characters (X'15').
14	+52	(See CMDSIZE)	<p>Address of the doubleword-aligned command table entry block (CMDBK) for the user-supplied base CP command. That is, if the user enters a command, this is the CMDBK for that command. If the user enters an alias, this is the CMDBK for the CP command to which the alias points, not the CMDBK for the alias.</p> <p>Note: You can find the length of the CMDBK in the CMDSIZE equate. The size value is stored in doublewords, not bytes.</p>
15	+56	(Stored in GSDFRESZ)	<p>Address of the fullword-aligned general system data block (GSDBK) for the user-supplied base CP command.</p> <p>Note: You can find the length of the GSDBK stored in the GSDFRESZ field. The size value is stored in doublewords, not bytes.</p>
16	+60	4 bytes	Address of the return code (supplied by this CP exit routine) that CP will use instead of mainline return code 20 or 24.

Exit conditions

On return, CP Exit OFFB sets the standard register contents described in [“Standard Exit Conditions” on page 94](#)

Return codes

On return, R15 contains two return codes. The first half of the fullword contains the "CP exit control return code", which is explained in [“Standard Return Codes” on page 94](#). The second half of the fullword contains the "mainline return code", which is explained in the following table:

Return Code	Results
0	CP ignores any changes made by this CP exit routine and continues normal processing.
4	<p>CP replaces the user-supplied command operands with the new CP exit-supplied operands described by Words 11, 12, and 13. CP does not change the name of the user-supplied CP command or subcommand.</p> <p>The contents of the field located by Word 12 determine which length (with or without logical line-end characters) CP will use:</p> <ul style="list-style-type: none">• If the value of the length field located by Word 12 is not zero, then CP replaces the command operands and all subsequent logical lines with the contents of the area located by Words 12 and 13.• Otherwise, CP replaces the command operands up to the end of the logical line with the contents of the area located by Words 11 and 13.
8	<p>CP replaces the command and its operands with the new data located by Words 11, 12, and 13. If this is a multiple-line command (more than one command strung together using logical line-end characters, X'15'), then CP performs another command authorization operation using the results of the replacements.</p> <ul style="list-style-type: none">• If the value of the length field located by Word 12 is not zero, then CP replaces the command and its operands and all subsequent logical lines with the contents of the area located by Words 12 and 13.• Otherwise, CP replaces the command and its operands up to the end of the logical line with the contents of the area located by Words 11 and 13.
12	CP ignores (does not process) the command by skipping to the next logical line. If the original command was multiple lines (for example, a SET PFnn command), CP will skip over only the first logical line. CP processes the remainder of the multiple-line input as CP commands.
16	CP ignores (does not process) the command and skips any remaining CP commands in a multiple-line command.
20	<p>CP rejects the command and skips to the next logical line. If the original command was a multiple-line command (for example, a SET PFnn command), CP only skips over the first logical line. CP processes the remainder of the multiple-line input as CP commands.</p> <p>CP sets the mainline return code to the value located by Word 16. This value is supplied by your CP exit routine.</p>
24	<p>CP rejects the command and skips any remaining commands. After rejecting the command, CP skips any remaining commands.</p> <p>CP sets the mainline return code to the value located by Word 16. This value is supplied by your CP exit routine.</p>

Return Code	Results
<i>nn</i>	For any other return codes, CP: <ul style="list-style-type: none"> • Writes error message HCP2765E to the system operator • Generates soft abend ABR002 • Ignores any changes made by this CP exit routine and continues normal processing.

Programming considerations

- You cannot bypass command and operand authorization checking using CP Exit OFFB.

CP Exit 1100: LOGON Command Pre-Parse Processing

Purpose

Use CP Exit 1100 to examine and, optionally, change or reject the CP LOGON command before CP parses the command.

Point of processing

Process	CP Exit Attribute
LOGON Command Pre-Parse Processing	<ul style="list-style-type: none"> • Reentrant • non-MP • No locks held

CP Exit 1100 is called immediately after a user enters the CP LOGON command, but before CP parses the command. You can use this CP exit point to:

- Specify CP LOGON command operands to:
 - Be added to the list of user-specified LOGON operands
 - Replace the list of user-specified LOGON operands
- Replace the specified console input data
- Reject the LOGON request.

CP provides a data area where the CP exit routine can examine the command operands and the console input data that were specified on the CP LOGON command. CP also provides data areas where the CP exit routine can store information for CP to use upon return.

Entry conditions

Register 1 points to a parameter list which is described below. For more information about other register contents, see [“Standard Entry Conditions” on page 93](#).

Parameter list constant

The parameter list contains the following:

Word	Offset	Data Length	Contents
1	+0	N/A	Reserved for future IBM use.

Word	Offset	Data Length	Contents
2	+4	32 bytes	Address of the doubleword-aligned User Area, which can be used by this CP exit routine as working storage.
3	+8	256 bytes	Address of the doubleword-aligned translate-and-test (TRT) table, which can be used by this CP exit routine as working storage.
4	+12	4 bytes	Address of the length of the command operands that were specified by the user on the CP LOGON command. The command operands are stored in the area located by Word 6. This does not include the length of the logical line-end character (X'15'), which marks the start of the console input data or the length of the console input data, if any.
5	+16	4 bytes	Address of the total length of the user-specified command operands and console input data area located by Word 6. This includes the length of all logical line-end characters (X'15'), which mark the start of the console input data and the length of the console input data, if any.
6	+20	(See Word 5)	Address of the area containing the command operands and console input data specified by the user on the CP LOGON command.
7	+24	4 bytes	Address of the maximum length of the area that this CP exit routine can use to store additional command operands or to replace the command operands, located by Word 9.
8	+28	4 bytes	Address of the length of the data that this CP exit routine stored in the area located by Word 9. CP initializes this length to zero to indicate that there is no data stored in the area located by Word 9. If your CP exit routine stores data in the area located by Word 9, your CP exit routine must update this length.
9	+32	(See Word 7)	Address of the CP exit-specified command operand area. Your CP exit routine can store additional or replacement LOGON command operands in this area.
10	+36	N/A	Address of general system data block (GSDBK) for the CP LOGON command.
11	+40	N/A	Address of the logon work buffer (LGNBK). The LGNBK contains flags, work areas, and data that describe the characteristics of the logon process. See to HCPLGNBK COPY for a description of the work area and its contents.
12	+44	N/A	Address of the virtual machine definition block (VMDBK) that CP created for this logon request. Because the user has not yet logged on, CP assigns a temporary user ID (such as LOGL0024).
13	+48	4 bytes	Address of the VMDRTERM field, which points to the real device control block (RDEV) of the display station from which the user issued the CP LOGON command. If there is no RDEV, this field will be 0 (zero).

Word	Offset	Data Length	Contents
14	+52	4 bytes	Address of the return code that CP should set if this CP exit routine rejects the CP LOGON command. CP sets the initial value of this field to 0 (zero). If rejecting the CP LOGON command, your CP exit routine must update this field. CP only uses this return code value if your CP exit routine completes with a return code that indicates the CP LOGON command is to be rejected.

Exit conditions

On return, CP Exit 1100 sets the standard register contents described in [“Standard Exit Conditions” on page 94](#).

Return codes

On return, R15 contains two return codes. The first half of the fullword contains the "CP exit control return code", which is explained in [“Standard Return Codes” on page 94](#). The second half of the fullword contains the "mainline return code", which is explained in the following table:

Return Code	Results
0	CP ignores any changes made by this CP exit routine and continues normal processing.
4	CP adds the CP exit-supplied command operands (Word 9) to the end of the user-supplied command operands, before any console input data (which appears after the logical line-end of the LOGON command).
8	CP replaces the user-specified command operands with the CP exit-supplied command operands (Word 9), but does not change the console input data (which appears after the logical line-end of the LOGON command).
12	CP replaces the user-specified command operands and any console input data with the CP exit-supplied command operands and console input data (Word 9).
16	CP rejects the command, using the CP exit-supplied return code (Word 14).
<i>nn</i>	For any other return codes, CP: <ul style="list-style-type: none"> • Writes error message HCP2765E to the system operator • Generates soft abend ABR002 • Ignores any changes made by this CP exit routine and continues normal processing.

Programming considerations

1. The address of the VMDRTERM field is passed in the parameter list because the value in this field can change during a loss of control as the result of switching to a different RDEV or loss of the RDEV. Your CP exit routine must validate the contents of this field before using them. If the field contains a zero, there is no RDEV associated with the VMDBK.
2. Your CP exit routine can only change the following parameter list fields:
 - The work areas located by Word 2 and Word 3.
 - The length value located by Word 8.
 - The values stored in the area located by Word 9.
 - The return code value located by Word 14.

You cannot make changes to any of the other parameter list fields.

CP Exit 1101: Logon Post-Parse Processing

Purpose

Use CP Exit 1101 to examine and, optionally, change the return code generated by CP after parsing the CP LOGON command.

Point of processing

Process	CP Exit Attribute
Logon Post-Parse Processing	<ul style="list-style-type: none">• Reentrant• non-MP• No locks held

CP Exit 1101 is called after CP parses a CP LOGON command, but before CP examines the return code from the parse (which CP uses to direct the rest of the logon process). You can use this CP exit point to:

- Examine the return code from the parse
- Change the return code from the parse.

Entry conditions

Register 1 points to a parameter list which is described below. For more information about other register contents, see [“Standard Entry Conditions”](#) on page 93.

Parameter list constant

The parameter list contains the following:

Word	Offset	Data Length	Contents
1	+0	N/A	Reserved for future IBM use.
2	+4	32 bytes	Address of the doubleword-aligned User Area, which can be used by this CP exit routine as working storage.
3	+8	256 bytes	Address of the doubleword-aligned translate-and-test (TRT) table, which can be used by this CP exit routine as working storage.
4	+12	N/A	Address of the LOGON work area (LGNBK), which contains flags, work areas, and data that describe the characteristics of the logon process. Refer to HCPLGNBK COPY for a description of the work area and its contents.
5	+16	N/A	Address of the virtual machine definition block (VMDBK) for the logging-on user. Because the user has not yet logged on, the user ID is a temporary one, such as LOGL0024, that CP assigns to a user in this state.
6	+20	4 bytes	Address of the VMDRTERM field which points to the real device control block (RDEV) of the console for the user who is logging on. The field will be zero if there is no RDEV.

Word	Offset	Data Length	Contents
7	+24	4 bytes	Address of the return code from the parse. On entry to your CP exit routine, the parse return code is the value returned by the standard parsing of the LOGON command. On exit from your CP exit routine, the parse return code is restricted to 0, 1, or 2. This value can be changed to control subsequent processing. A parse return code value of 0 tells CP to continue processing the LOGON command. A return code value of 1 tells CP to terminate processing of the LOGON command. A return code value of 2 tells CP to perform a forced logon of the system operator. Any other value is ignored and LOGON processing continues based on the original parse return code.

Exit conditions

On return, CP Exit 1101 sets the standard register contents described in [“Standard Exit Conditions” on page 94](#).

Return codes

On return, R15 contains two return codes. The first half of the fullword contains the "CP exit control return code", which is explained in [“Standard Return Codes” on page 94](#). The second half of the fullword contains the "mainline return code", which is explained in the following table:

Return Code	Results
0	CP validates the parse return code located by Word 7. Valid parse return codes are: <ul style="list-style-type: none"> 0 tells CP to continue processing the LOGON command. 1 tells CP to terminate processing of the LOGON command. 2 tells CP to perform a forced logon of the system operator. <p>If the parse return code (Word 7) is valid, CP continues LOGON processing based on the value of the parse return code. For any other parse return code value (Word 7), CP:</p> <ul style="list-style-type: none"> • Writes error message HCP2765E to the system operator • Generates soft abend ABR002 • Ignores any changes made by this CP exit routine and continues normal processing of the LOGON command based on the value of the original parse return code.
<i>nn</i>	For any other return codes, CP: <ul style="list-style-type: none"> • Writes error message HCP2765E to the system operator • Generates soft abend ABR002 • Ignores any changes made by this CP exit routine and continues normal processing of the LOGON command based on the value of the original parse return code.

Programming considerations

1. The address of the VMDRTERM field is passed in the parameter list because the value in this field can change during a loss of control as the result of switching to a different RDEV or loss of the RDEV. Your CP exit routine must validate the contents of this field before using them. If the field contains a zero, there is no RDEV associated with the VMDBK.
2. Your CP exit routine can only change the following parameter list fields:

The work areas located by Word 2 and Word 3.

The return code value located by Word 7.

You cannot make changes to any of the other parameter list fields.

CP Exit 1110: VMDBK Pre-Logon Processing

Purpose

Use CP Exit 1110 to examine and, optionally, change the virtual machine definition block (VMDBK) for a virtual machine after CP creates and initializes it.

Point of processing

Process	CP Exit Attribute
VMDBK Pre-Logon Processing	<ul style="list-style-type: none">• Reentrant• non-MP• No locks held

CP Exit 1110 is called during virtual machine creation after CP has:

- Created and initialized the VMDBK,
- Defined all the virtual CPUs,
- Established all the virtual devices, and
- Created the base address space.

CP provides the CP exit routine with the address of the LOGON work area (described by HCPLGNBK COPY), which indicates the source of the virtual machine creation (for example, the CP AUTOLOG command), and provides other information about the process.

Entry conditions

Register 1 points to a parameter list which is described below. For more information about other register contents, see [“Standard Entry Conditions”](#) on page 93.

Parameter list constant

The parameter list contains the following:

Word	Offset	Data Length	Contents
1	+0	N/A	Reserved for future IBM use.
2	+4	32 bytes	Address of the doubleword-aligned User Area, which can be used by this CP exit routine as working storage.
3	+8	256 bytes	Address of a doubleword-aligned translate-and-test (TRT) table, which can be used by this CP exit routine as working storage.
4	+12	8 bytes	Address of the 8-byte area containing the user ID of the logging-on user.

Word	Offset	Data Length	Contents
5	+16	N/A	Address of the LOGON work area (LGNBK), which contains flags, work areas, and data that describe the characteristics of the logon process. At the point this CP exit point is called, the LGNDVMD field is valid and the data structures needed to read the User Directory are set up. Refer to HCPLGNBK COPY for a description of the work area and its contents.
6	+20	N/A	Address of the virtual machine definition block (VMDBK) for the logging-on user. Because the user has not yet logged on, the user identifier is a temporary one, such as LOGL0024, that CP assigns to a user in this state.
7	+24	4 bytes	Address of the VMDRTERM field, which points to the real device control block (RDEV) of the console for the logging-on user. The field will be zero if there is no real device control block (RDEV).

Exit conditions

On return, CP Exit 1110 sets the standard register contents described in [“Standard Exit Conditions” on page 94](#).

Return codes

On return, R15 contains two return codes. The first half of the fullword contains the "CP exit control return code", which is explained in [“Standard Return Codes” on page 94](#). The second half of the fullword contains the "mainline return code", which is explained in the following table:

Return Code	Results
0	CP continues normal processing.
<i>nn</i>	For any other return codes, CP: <ul style="list-style-type: none"> Writes error message HCP2765E to the system operator Generates soft abend ABR002 Continues normal processing.

Programming considerations

- The address of the VMDRTERM field is passed in the parameter list because the value in this field can change during a loss of control as the result of switching to a different RDEV or loss of the RDEV. Your CP exit routine must validate the contents of this field before using them. If the field contains a zero, there is no RDEV associated with the VMDBK.
- Your CP exit routine can only change the following parameter list fields:
 - The work areas located by Word 2 and 3
 - The VMDBK located by Word 6.

You cannot make changes to any of the other parameter list fields.

Important Note: Although you can use CP Exit 1110 to change the VMDBK, we must caution you that any such modifications can result in unpredictable side effects. Locating and correcting these side effects is the responsibility of the writer of the CP exit routine.

CP Exit 117F: Logon Final Screening

Purpose

Use CP Exit 117F to examine and, optionally, perform any processing required immediately before a successful LOGON command completes.

Point of processing

Process	CP Exit Attribute
Logon Final Screening	<ul style="list-style-type: none">• Reentrant• non-MP• No locks held

CP Exit 117F is called after CP completes all logon processing, but immediately before terminating the execution of the LOGON command. CP provides the CP exit routine with the address of the LOGON work area (described by HCPLGNBK COPY), which indicates the source of the logon activity (for example, the CP AUTOLOG command), and provides other information about the process.

Entry conditions

Register 1 points to a parameter list which is described below. For more information about other register contents, see [“Standard Entry Conditions” on page 93](#).

Parameter list constant

The parameter list contains the following:

Word	Offset	Data Length	Contents
1	+0	N/A	Reserved for future IBM use.
2	+4	32 bytes	Address of the doubleword-aligned User Area, which can be used by this CP exit routine as working storage.
3	+8	256 bytes	Address of the doubleword-aligned translate-and-test (TRT) table, which can be used by this CP exit routine as working storage.
4	+12	8 bytes	Address of the 8-byte area containing the user ID of the logging-on user.
5	+16	N/A	Address of the LOGON work area (LGNBK), which contains flags, work areas, and data that describe the characteristics of the logon process. Refer to HCPLGNBK COPY for a description of the work area and its contents.
6	+20	N/A	Address of the virtual machine definition block (VMDBK) for the logging-on user.
7	+24	4 bytes	Address of the VMDRTERM field which points to the real device control block (RDEV) of the console for the logging-on user. The field will be zero if there is no real device control block (RDEV).

Exit conditions

On return, CP Exit 117F sets the standard register contents described in [“Standard Exit Conditions” on page 94](#).

Return codes

On return, R15 contains two return codes. The first half of the fullword contains the "CP exit control return code", which is explained in ["Standard Return Codes" on page 94](#). The second half of the fullword contains the "mainline return code", which is explained in the following table:

Return Code	Results
0	CP continues normal processing.
<i>nn</i>	For any other return codes, CP: <ul style="list-style-type: none">• Writes error message HCP2765E to the system operator• Generates soft abend ABR002• Continues normal processing.

Programming considerations

1. The address of the VMDRTERM field is passed in the parameter list because the value in this field can change during a loss of control as the result of switching to a different RDEV or loss of the RDEV. Your CP exit routine must validate the contents of this field before using them. If the field contains a zero, there is no RDEV associated with the VMDBK.
2. Your CP exit routine can only change the following parameter list fields:
 - The work areas located by Word 2 and Word 3.
 - You cannot make changes to any of the other parameter list fields.

CP Exit 11C0: Logoff Initial Screening

Purpose

Use CP Exit 11C0 to examine the CP LOGOFF command immediately after it is issued.

Point of processing

Process	CP Exit Attribute
Logoff Initial Screening	<ul style="list-style-type: none">• Reentrant• non-MP• No locks held

CP Exit 11C0 is called immediately after a CP LOGOFF command is issued and CP has set the appropriate status flags in the VMDBK, but before CP has:

- Displayed the LOGOFF message,
- Terminated console spooling,
- Performed any other cleanup processing associated with logging off a virtual machine.

Entry conditions

Register 1 points to a parameter list which is described below. For more information about other register contents, see ["Standard Entry Conditions" on page 93](#).

Parameter list constant

The parameter list contains the following:

Word	Offset	Data Length	Contents
1	+0	N/A	Reserved for future IBM use.
2	+4	32 bytes	Address of the doubleword-aligned User Area, which can be used by this CP exit routine as working storage.
3	+8	256 bytes	Address of the doubleword-aligned translate-and-test (TRT) table, which can be used by this CP exit routine as working storage.
4	+12	8 bytes	Address of the 8-byte area containing the user ID of the logging-off user.
5	+16	N/A	Address of the LOGOFF work area (LGFBK), which contains flags, work areas, and data that describe the characteristics of the logoff process. Refer to HCPLGFBK COPY for a description of the work area and its contents. The work area may not exist, in which case this parameter will be zero. If the logoff is the result of a LOGOFF command or a forced disconnect, the work area will exist.
6	+20	1 byte	Address of the VMATYPE field, which identifies the type of VMDBK.
7	+24	N/A	Address of the virtual machine definition block (VMDBK) for the logging-off user.
8	+28	4 bytes	Address of the VMDRTERM field which points to the real device control block (RDEV) of the console for the logging-off user. The field will be zero if there is no real device control block (RDEV).

Exit conditions

On return, CP Exit 11C0 sets the standard register contents described in [“Standard Exit Conditions” on page 94](#).

Return codes

On return, R15 contains two return codes. The first half of the fullword contains the "CP exit control return code", which is explained in [“Standard Return Codes” on page 94](#). The second half of the fullword contains the "mainline return code", which is ignored.

Programming considerations

1. The address of the VMDRTERM field is passed in the parameter list because the value in this field can change during a loss of control as the result of switching to a different RDEV or loss of the RDEV. Your CP exit routine must validate the contents of this field before using them. If the field contains a zero, there is no RDEV associated with the VMDBK.
2. Your CP exit routine can only change the following parameter list fields:

The work areas located by Word 2 and Word 3.

You cannot make changes to any of the other parameter list fields.

CP Exit 11FF: Logoff Final Screening

Purpose

Use CP Exit 11FF to examine and, optionally, perform any processing required immediately before a successful LOGOFF command completes.

Point of processing

Process	CP Exit Attribute
Logoff Final Screening	<ul style="list-style-type: none">• Reentrant• non-MP• No locks held

CP Exit 11FF is called after CP completes most logoff processing, but before CP terminates the execution of the LOGOFF command. At this point, the virtual machine definition block (VMDBK) still exists, but CP has released most of the data areas to which it referred.

Entry conditions

Register 1 points to a parameter list which is described below. For more information about other register contents, see [“Standard Entry Conditions” on page 93](#).

Parameter list constant

The parameter list contains the following:

Word	Offset	Data Length	Contents
1	+0	N/A	Reserved for future IBM use.
2	+4	32 bytes	Address of the doubleword-aligned User Area, which can be used by this CP exit routine as working storage.
3	+8	256 bytes	Address of the doubleword-aligned translate-and-test (TRT) table, which can be used by this CP exit routine as working storage.
4	+12	8 bytes	Address of the 8-byte area containing the user ID of the logging-off user.
5	+16	1 byte	Address of VMTYPE field, which identifies the type of virtual machine definition block (VMDBK).
6	+20	N/A	Address of the virtual machine definition block (VMDBK) for the logging-off user.

Exit conditions

On return, CP Exit 11FF sets the standard register contents described in [“Standard Exit Conditions” on page 94](#).

Return codes

On return, R15 contains two return codes. The first half of the fullword contains the "CP exit control return code", which is explained in [“Standard Return Codes” on page 94](#). The second half of the fullword contains the "mainline return code", which is ignored.

Programming considerations

1. Your CP exit routine can only change the following parameter list fields:

The work areas located by Word 2 and Word 3.

You cannot make changes to any of the other parameter list fields.

CP Exit 1200: DIAL Command Initial Screening

Purpose

Use CP Exit 1200 to examine and, optionally, change or reject the operands specified on the CP DIAL command.

Point of processing

Process	CP Exit Attribute
DIAL Command Initial Screening	<ul style="list-style-type: none">• Reentrant• MP• No locks held

CP Exit 1200 is called immediately after a user enters the CP DIAL command and CP has parsed its operands but before CP processes them. You can use this CP exit point to examine, modify, or reject the DIAL request.

Entry conditions

Register 1 points to a parameter list which is described below. For more information about other register contents, see [“Standard Entry Conditions” on page 93](#).

Parameter list constant

The parameter list contains the following:

Word	Offset	Data Length	Contents
1	+0	4 bytes	Reserved for future IBM use.
2	+4	32 bytes	Address of the doubleword-aligned user area, which can be used by this CP exit routine as working storage.
3	+8	256 bytes	Address of the doubleword-aligned translate-and-test (TRT) table, which can be used by this CP exit routine as working storage.
4	+12	12 bytes	Address of the command name as entered by the user, left justified and padded with blanks.
5	+16	8 bytes	Address of the target user ID as entered by the user, left justified and padded with blanks. If no target user ID was entered, then this field will be all blanks.
6	+20	4 bytes	Address of the target virtual device number as entered by the user, after conversion to binary. If no target virtual device number was entered, then this field will be all 1 bits.

Word	Offset	Data Length	Contents
7	+24	4 bytes	Address of the virtual device number where CP should start searching for a target virtual device number. This field is initialized to binary 0.
8	+28	4 bytes	Address of the virtual device number where CP should stop searching for a target virtual device number. This field is initialized to X'FFFF'.
9	+32	N/A	Address of the RDEV control block for the terminal where the user issued the command.

Exit conditions

On return, CP Exit 1200 sets the standard register contents described in [“Standard Exit Conditions” on page 94](#).

Return codes

On return, R15 contains two return codes. The first half of the fullword contains the "CP exit control return code", which is explained in [“Standard Return Codes” on page 94](#). The second half of the fullword contains the "mainline return code", which is explained in the following table:

Return Code	Results
0	CP continues normal processing, using the initial values or the values replaced by the CP exit routine.
4	CP: <ul style="list-style-type: none"> Writes message HCP2779E to the system operator Rejects the command.
8	CP: <ul style="list-style-type: none"> Does not writes any messages Rejects the command.
<i>nn</i>	For any other return codes, CP: <ul style="list-style-type: none"> Writes message HCP2765E to the system operator Generates soft abend ABR002 Rejects the DIAL command.

Programming considerations

1. CP Exit 1200 routines can change the following parameter list fields:

- The target user ID pointed to by Word 5
- The target virtual device number pointed to by Word 6
- The starting virtual device number for searching pointed to by Word 7
- The ending virtual device number for searching pointed to by Word 8.

You cannot make changes to the other parameter list fields.

2. A virtual device number must be in the range X'0000' through X'FFFF'. If the CP exit routine returns a virtual device number that would be illegal, CP will treat this as if a return code of 4 was returned from the CP exit routine.

CP Exit 1201: DIAL Command Final Screening

Purpose

Use CP Exit 1201 to examine and, optionally, reject the decisions made by the processing of the CP DIAL command.

Point of processing

Process	CP Exit Attribute
DIAL Command Final Screening	<ul style="list-style-type: none">• Reentrant• MP• VDEV lock for the target user ID's virtual device control block (VDEV) selected to satisfy that the request is held, and must not be relinquished at any time.

CP Exit 1201 is called immediately after CP has processed the DIAL command operands and knows the target user ID and the target virtual device number, but before CP completes the actual connection. Using this CP exit point, you can only examine the decisions and their values and choose whether to accept this DIAL command or to reject it. You cannot use this CP exit point to change any of the decisions or their values.

Entry conditions

Register 1 points to a parameter list which is described below. For more information about other register contents, see [“Standard Entry Conditions” on page 93](#).

Parameter list constant

The parameter list contains the following:

Word	Offset	Data Length	Contents
1	+0	4 bytes	Reserved for future IBM use.
2	+4	32 bytes	Address of the doubleword-aligned user area, which can be used by this CP exit routine as working storage.
3	+8	256 bytes	Address of the doubleword-aligned translate-and-test (TRT) table, which can be used by this CP exit routine as working storage.
4	+12	8 bytes	Address of the target user ID, left justified and padded with blanks.
5	+16	N/A	Address of the VDEV control block belonging to the target user ID chosen by CP to satisfy the request.
6	+20	N/A	Address of the real device control block (RDEV) for the terminal where the user issued the command.

Exit conditions

On return, CP Exit 1201 sets the standard register contents described in [“Standard Exit Conditions” on page 94](#).

Return codes

On return, R15 contains two return codes. The first half of the fullword contains the "CP exit control return code", which is explained in [“Standard Return Codes” on page 94](#). The second half of the fullword contains the "mainline return code", which is explained in the following table:

Return Code	Results
0	CP continues normal processing, using the initial values or the values set by CP Exit 1200.
<i>nn</i>	For any other return codes, CP: <ul style="list-style-type: none">• Writes message HCP2779E to the operator• Rejects the DIAL command.

Programming considerations

1. CP Exit 1201 routines cannot change what has been decided. The CP exit routines can only accept or reject the final decisions about the target user ID and the target virtual device number.

CP Exit 1210: Messaging Commands Screening

Purpose

Use CP Exit 1210 to examine and, optionally, change or reject one of the following messaging commands: MESSAGE, MSGNOH, SMSG, and WARNING. You can:

- Add more verification
- Change the format of the message
- Change the context of the message
- Change the parameters that control the way the message is displayed on the screen
- Process installation-defined message command options.

Point of processing

Process	CP Exit Attribute
Messaging Commands Screening	<ul style="list-style-type: none">• Reentrant• MP• No locks held

CP Exit 1210 is called after a user issues one of the messaging commands (MESSAGE, MSGNOH, SMSG, or WARNING), but before CP sends the formatted message to the designated receiver (user ID).

Entry conditions

Register 1 points to a parameter list which is described below.

Register 11 contains the address of the message sender's VMDBK. For more information about other register contents, see [“Standard Entry Conditions”](#) on page 93.

Parameter list constant

The parameter list contains the following:

Word	Offset	Data Length	Contents
1	+0	N/A	Reserved for future IBM use.
2	+4	32 bytes	Address of the doubleword-aligned user area, which can be used by this CP exit routine as working storage.
3	+8	256 bytes	Address of the doubleword-aligned translate-and-test (TRT) table, which can be used by this CP exit routine as working storage.

Word	Offset	Data Length	Contents
4	+12	32 bits	<p>Address of the 32 bit message control flags, which contain flag bits that control the way the message is displayed. The first 8 bits of this 32-bit field control the way the message is displayed. More than one bit may be set. You can change the settings to change the way the message is displayed. The possible values are:</p> <p>Value Meaning</p> <p>X'80_____' A time stamp is appended to the message.</p> <p>X'40_____' No time stamp is appended to the message.</p> <p>X'20_____' The message is highlighted when displayed.</p> <p>X'10_____' The console alarm sounds when the message is displayed.</p> <p>X'08_____' This is a high-priority message and is displayed immediately.</p> <p>X'04_____'-X'_____01' Reserved for IBM use.</p> <p>Note: If neither of the first two bits which control the time stamp are set, the receiving user's value for the VMDTSTAM bit in the VMDTOPTN field of the VMDBK is used. If both bits are set, a time stamp is appended.</p> <p>The supplied display values for the message commands are:</p> <p>Command/Value Meaning</p> <p>MESSAGE/X'B0' Time stamp is appended; message is highlighted; console alarm sounds.</p> <p>MSGNOH/X'70' No time stamp is appended; message is highlighted; console alarm sounds.</p> <p>SMSG/X'00' This type of message is not displayed.</p> <p>WARNING/X'B8' Time stamp is appended; message is highlighted; console alarm sounds; message is high-priority, displayed immediately.</p>

Word	Offset	Data Length	Contents
5	+16	32 bits	<p>Address of the 32 bit field containing the message type and Issuer IBM class field. This 32-bit field consists of three subfields:</p> <ol style="list-style-type: none"> 1. The first 8 bits indicate the type of message being processed. You can change the setting to change the way the message is processed. Note that if you change this setting, you might also need to change other parameters. The possible values are: <ul style="list-style-type: none"> Value: Meaning: X'80' - MESSAGE command processing X'40' - MSGNOH command processing X'20' - SMSG command processing X'10' - WARNING command processing X'08'–X'01' - Reserved for IBM use 2. The next 8 bits indicate which IBM class commands the issuer of the message is allowed to execute. More than one bit may be set. The settings are dependent on the privilege class of the issuer of the message as well as the privilege class of the command entered. This field is included for reference only. Changes made to this field do not affect the way the command is processed. The possible values are: <ul style="list-style-type: none"> Value: Meaning: X'80' - IBM class A commands X'40' - IBM class B commands X'20' - IBM class C commands X'10' - IBM class D commands X'08' - IBM class E commands X'04' - IBM class F commands X'02' - IBM class G commands X'01' - IBM class H commands X'00' - IBM class ANY 3. The final 16 bits are reserved for IBM use.
6	+20	8 bytes	<p>Address of the 8 byte field containing the user ID of the issuer of the command. This field is included for reference only. Changes made to this field do not affect the way the command is processed.</p>
7	+24	32 bits	<p>Address of the 32 bit field containing the length of the message being processed. Changes to this field might affect the way the message is displayed.</p>

Word	Offset	Data Length	Contents
8	+28	4 bytes	<p>Address of the address of the message being processed. This 32-bit field contains the address of the message being processed. For each type of message, the message buffer looks as follows:</p> <p>Message Type Buffer Contents</p> <p>MESSAGE * MSG FROM <i>userid</i> : <i>text</i></p> <p>MSGNOH <i>text</i></p> <p>SMSG <i>text</i></p> <p>WARNING * WNG FROM <i>userid</i> : <i>text</i></p> <p>Note: The MESSAGE and WARNING messages always contain an initial blank and provide eight spaces for the user ID before the colon, regardless of the actual length of the user ID.</p> <p>Following the text is 100 bytes of unused space which can be used to add to or modify the existing text. If the length of the message is changed, the length field described above should be changed to reflect this or not all text will be displayed.</p> <p>You can change the address of the message (and also the length of the message, if needed) for such purposes as, for example, pointing around the message header so it is not displayed. However, when the CP message function releases the storage it uses for the message, it releases the storage based on the address passed to the exit routine, not the updated value.</p>
9	+32	8 bytes	<p>Address of the 8 byte field containing the user ID of the receiver of the command. If the value of the receiver's user ID is changed, the message is sent to the new receiver. However, if an error message is issued during normal message-function processing, and the error message includes the receiver's user ID, the user ID used in the error message is the one that was passed to the exit routine, not the changed value.</p>

Word	Offset	Data Length	Contents
10	+36	4 bytes	<p>Address of a 32 bit field which contains the length of the header for the message being processed. Each message is made up of a header and a text portion. The header includes the header information and the blank delimiter which precedes the text.</p> <p>The header length is used by the message processing function when messages are sent across the *MSG, *MSGALL, and VMCF services. Only the text of the message is sent. The header is discarded. The header length allows the message function to determine where the message text begins.</p> <p>Each message type is considered to have a header. For MESSAGE and WARNING types, the header length on entry to the exit routine is 22 characters long. For MSGNOH and SMSG, the header length on entry to the exit routine is 0 characters long.</p> <p>The header can be increased or decreased by this exit or by later message processing (for example, when a time stamp is added). SMSG never transmits the header while the other types display a header at the terminal if one is present.</p>

Exit conditions

On return, CP Exit 1210 sets the standard register contents described in [“Standard Exit Conditions” on page 94](#).

Return codes

On return, R15 contains two return codes. The first half of the fullword contains the "CP exit control return code", which is explained in [“Standard Return Codes” on page 94](#). The second half of the fullword contains the "mainline return code", which is explained in the following table:

Return Code	Results
0	CP continues normal processing and ignores any changes made by this CP exit routine.
<i>nnnn</i>	<p>CP stops processing the message. The installation-defined return code <i>nnnn</i> can be in the range of 0001 to 270F (decimal 9999). The following error message will be issued to the invoker of the message function:</p> <pre>HCPMFS6600E An error was detected by installation-wide exit point 1210 – return code <i>mmmm</i>.</pre> <p>where <i>mmmm</i> is the decimal value of the return code. See <i>z/VM: CP Messages and Codes</i> for further information on this message. For any return code greater than 270F (decimal 9999), CP:</p> <ul style="list-style-type: none"> Writes error message HCP2765.01E to the operator. Writes error message HCP6600.02E to the invoker of the command passing back the last four decimal digits of the return code set by the exit routine.

Programming considerations

None.

Migration aids

Previous versions of VM let you perform a similar function by adding source code to module HCPMSU for entry point HCPMSUEX as described in [z/VM: CP Planning and Administration](#). CP Exit 1210 is intended to replace HCPMSUEX. If routine associated with CP Exit 1210 is executed then HCPMSUEX is not called.

CP Exit 1230: VMRELOCATE Eligibility Checks

Purpose

Use CP Exit 1230 to define installation-specific conditions for disallowing the relocation of virtual machines using the VMRELOCATE command in a single system image cluster.

Point of processing

Process	CP Exit Attribute
VMRELOCATE Eligibility Checks	<ul style="list-style-type: none">• Reentrant• MP• No locks held

CP Exit 1230 is called twice during the processing of a VMRELOCATE command. The first time is early in command processing to test the eligibility for relocation of the virtual machine to the specified destination system. The second time is after the virtual machine is stopped, to ensure no changes made to the virtual machine during the early part of the relocation (before the virtual machine is stopped) have made the virtual machine ineligible.

Entry conditions

Register 1 points to a parameter list which is described below. For more information about other register contents, see [“Standard Entry Conditions”](#) on page 93.

Parameter list constant

The parameter list, which is defined as X1230 dsect in HCPPLXIT COPY, contains the following:

Word	Offset	Data Length	Contents
1	+0	N/A	Reserved for future IBM use.
2	+4	32 bytes	Address of the doubleword-aligned user area, which can be used by this CP exit routine as working storage.
3	+8	256 bytes	Address of the doubleword-aligned translate-and-test (TRT) table, which can be used by this CP exit routine as working storage.
4	+12	RLOBSIZE	Host logical address of the relocation control block (RLOBK dsect defined in HCPRLOBK COPY). The RLOBK contains information about the relocation such as: target user, destination system, and a flag indicating if this is the first time (initial eligibility checks) or second time (after virtual machine is stopped) the exit is being called.

Exit conditions

On return, CP Exit 1230 sets the standard register contents described in [“Standard Exit Conditions” on page 94](#).

Return codes

On return, R15 contains two return codes. The first half of the fullword contains the "CP exit control return code", which is explained in [“Standard Return Codes” on page 94](#). The second half of the fullword contains the "mainline return code", which is explained in the following table:

Return Code	Results
0	The installation-supplied eligibility routine has determined that the relocation request may continue. The standard IBM relocation eligibility checks should be done and the relocation proceeds if no violations are detected. If an eligibility violation is detected, the relocation should be canceled.
4	The first time this exit is called during a relocation, the installation-supplied eligibility routine has determined that the relocation request should fail but all other eligibility checks should be made before terminating the command. This return code allows all eligibility violations to be listed in the command output. On the second call to the exit, this return Code has the same meaning as return code 8.
8	The installation-supplied eligibility routine has determined that the relocation request should be terminated immediately.

Programming considerations

1. The installation-supplied exit should display any messages necessary to indicate which installation-specific eligibility restrictions would be violated by this relocation request.

CP Exit 1231: VMRELOCATE Destination Restart

Purpose

Use CP Exit 1231 to do any necessary processing before the target guest is restarted on the destination system during a live guest relocation.

Point of processing

Process	CP Exit Attribute
VMRELOCATE Destination Restart	<ul style="list-style-type: none">• Reentrant• MP• No locks held

CP Exit 1231 is called once during the processing of a VMRELOCATE command. After the guest is temporarily halted and the guest state has been moved from the source to the destination system, this exit is called on the destination system prior to restart of the guest.

Entry conditions

Register 1 points to a parameter list which is described below. For more information about other register contents, see [“Standard Entry Conditions” on page 93](#).

Parameter list constant

The parameter list, which is defined as X1231 dsect in HCPPLXIT COPY, contains the following:

Word	Offset	Data Length	Contents
1	+0	N/A	Reserved for future IBM use.
2	+4	32 bytes	Address of the doubleword-aligned user area, which can be used by this CP exit routine as working storage.
3	+8	256 bytes	Address of the doubleword-aligned translate-and-test (TRT) table, which can be used by this CP exit routine as working storage.
4	+12	RLOBSIZE	Host logical address of the relocation control block (RLOBK dsect defined in HCPRLOBK COPY). The RLOBK contains information about the relocation such as: target user, destination system, VMRELOCATE command options, etc..

Exit conditions

On return, CP Exit 1231 sets the standard register contents described in [“Standard Exit Conditions”](#) on page 94.

Return codes

On return, R15 contains two return codes. The first half of the fullword contains the "CP exit control return code", which is explained in [“Standard Return Codes”](#) on page 94. The second half of the fullword contains the "mainline return code", which is explained in the following table:

Return Code	Results
0	Allow relocation to continue.
non-zero	Cancel the relocation.

Programming considerations

1. The installation-supplied exit should display any messages necessary to indicate why the relocation was rejected if a non-zero R15 (return code) is returned to the caller.
2. This exit is called during the time when the relocating guest is quiesced. It is critical that this quiesce time should be as short as possible.

CP Exit 3FE8: SHUTDOWN Command Screening

Purpose

Use CP Exit 3FE8 to

- examine the operands supplied on the SHUTDOWN
- add, remove, or alter any operand on the SHUTDOWN command
- accept or reject the SHUTDOWN command

Point of processing

Process	CP Exit Attribute
SHUTDOWN Command Screening	<ul style="list-style-type: none"> • Reentrant • MP • No locks held

CP Exit 3FE8 is called after a CP SHUTDOWN command has been issued and authorized but before CP has examined any operands of the command. The calling task provides to the exit routine

- standard exit work areas
- a copy of the operands from the command
- an area for the exit to provide replacement operands
- an area for the exit to provide a command return code if the exit routine rejects the command

Entry conditions

Register 1 points to a parameter list which is described below. For more information about other register contents, see [“Standard Entry Conditions” on page 93](#).

Parameter list constant

The parameter list contains the following:

Word	Offset	Data Length	Contents
1	+0	N/A	Reserved for future IBM use.
2	+4	32 bytes	Address of the doubleword-aligned user area, which may be used by each CP exit routine as working storage.
3	+8	256 bytes	Address of the doubleword-aligned translate-and-test (TRT) table, which may be used by each CP exit routine as working storage.
4	+12	4 bytes	Address of the length of the data located by Word 5.
5	+16	(See Word 4)	Address of a copy of the operands from the SHUTDOWN command.
6	+20	4 bytes	Address of the fullword-aligned maximum length of the data area located by Word 8.
7	+24	4 bytes	Address of the fullword-aligned length of the data in the area located by Word 8.
8	+28	(See Word 7)	Address of replacement operands for the SHUTDOWN command.
9	+32	4 bytes	Address of the command error return code.

Return codes

On return, R15 contains two return codes. The first half of the fullword contains the "CP exit control return code", which is explained in [“Standard Return Codes” on page 94](#). The second half of the fullword contains the "mainline return code", which is explained in the following table:

Return Code	Results
0	This return code indicates that no changes were made to the operands on the SHUTDOWN command. CP continues processing of the SHUTDOWN command.

Return Code	Results
4	This return code indicates that the exit has supplied additional command operands. These operands are returned by the exit routine using Word 7 (label X3FE8OUL in the exit parameter list DSECT) and Word 8 (label X3FE8OU in the exit parameter list DSECT). CP inserts these operands into the command immediately after the command and before the original operands. CP continues processing of the SHUTDOWN command using these resulting operands. The operands returned by the exit should begin with a blank character so that CP can perform proper parsing of the command operands.
8	This return code indicates that the exit has supplied replacement command operands. These operands are returned by the exit routine using Word 7 (label X3FE8OUL in the exit parameter list DSECT) and Word 8 (label X3FE8OU in the exit parameter list DSECT). CP inserts these operands into the command immediately after the command and before the original operands. The original command operands are discarded. CP continues processing of the SHUTDOWN command using these resulting operands. The operands returned by the exit should begin with a blank character so that CP can perform proper parsing of the command operands.
12	This return code indicates that the exit has rejected the SHUTDOWN command. CP will use as the SHUTDOWN command error return code the value located by Word 9 (label X3FE8RC in the exit parameter list DSECT). CP will not generate any additional error message. CP assumes that the exit routine has already generated any error messages that it deemed appropriate.
<i>nn</i>	For any other return codes, CP: <ul style="list-style-type: none"> • Writes message HCP2765E to the operator. • Generates a soft abend ABR002. • Continues normal processing as if return code 0 were returned.

Programming considerations

1. Unavailable CP services

- All CP services are available.

2. Your exit routines can change the following parameter list fields:

- The work areas located by Word 2 and Word 3.
- The length value located by Word 7.
- The values stored in the area located by Word 8.
- The command error return code value located by Word 9.

You cannot make changes to the other parameter list fields.

CP Exit 4400: Separator Page Data Customization

Purpose

Use CP Exit 4400 to examine and, optionally, modify the information that will be printed on the VM separator page for printed output.

Point of processing

Process	CP Exit Attribute
Separator Page Data Customization	<ul style="list-style-type: none"> • Reentrant • MP • No locks held

CP Exit 4400 is called during the processing of VM separator pages, after CP fills in the information that will be printed on the separator page, but before CP prints it. The separator page information is stored in buffers that are mapped by the SEPPAG1 and SEPPAG2 DSECTs.

Entry conditions

Register 1 points to a parameter list which is described below. For more information about other register contents, see [“Standard Entry Conditions” on page 93](#).

Parameter list constant

The parameter list contains the following:

Word	Offset	Length	Contents
1	+0	N/A	Reserved for future IBM use.
2	+4	32 bytes	Address of the doubleword-aligned user area, which is used by this CP exit routine as working storage.
3	+8	256-byte	Address of the doubleword-aligned translate-and-test (TRT) table, which is used by this CP exit routine as working storage.
4	+12	N/A	Reserved for future IBM use.
5	+16	4096 bytes	Address of the SEPPAG1 buffer which contains separator page channel command words (CCWs) and data.
6	+20	4096 bytes	Address of the SEPPAG2 buffer which contains separator page data.
7	+24	SPFEND (bytes)	Address of the spool file control block (SPFBK) of the file being printed.
8	+28	RDEVSZCK (doublewords)	Address of the real device control block (RDEV) of the printer that will print the file.
9	+32	RSPSIZE (doublewords)	Address of the real spool device block (RSPBK) of the printer that will print the file.

Exit conditions

On return, CP Exit 4400 sets the standard register contents described in [“Standard Exit Conditions” on page 94](#).

Return codes

On return, R15 contains two return codes. The first half of the fullword contains the "CP exit control return code", which is explained in [“Standard Return Codes” on page 94](#). The second half of the fullword contains the "mainline return code", which is explained in the following table:

Return Code	Results
0	CP continues normal processing.

Return Code	Results
4	CP continues normal processing.
8	CP terminates separator page processing for the current file.

Programming considerations

1. If you modify the contents of SEPPAG1 or SEPPAG2 DSECT, be aware that each file cannot exceed 4096 bytes in length.
2. To terminate the processing of this VM separator page, you do not have to create any dynamically loaded CP routines. All you need to do is:
 - Associate IBM-supplied entry point HCPSRC08 with this CP exit point using the ASSOCIATE EXIT command (or configuration file statement).
 - Enable this CP exit point using the ENABLE operand of ASSOCIATE EXIT or using the ENABLE EXITS command (or configuration file statement).

For more information on the ASSOCIATE EXIT or ENABLE EXITS commands, see [z/VM: CP Commands and Utilities Reference](#). For more information on the ASSOCIATE EXIT or ENABLE EXITS statements, see [z/VM: CP Planning and Administration](#).

CP Exit 4401: Separator Page Pre-Perforation Positioning

Purpose

Use CP Exit 4401 to change the channel program which positions impact printers at the bottom of a form for printing across the perforation.

Point of processing

Process	CP Exit Attribute
Separator Page Pre-Perforation Positioning	<ul style="list-style-type: none"> • Reentrant • MP • No locks held

CP Exit 4401 is called during separator page processing, prior to positioning the form on an impact printer for perforation printing. The channel program is mapped by SEPPAG1 DSECT beginning at label SEPCWP1 in HCPSEPBK COPY.

Entry conditions

Register 1 points to a parameter list which is described below. For more information about other register contents, see [“Standard Entry Conditions” on page 93](#).

Parameter list constant

The parameter list contains the following:

Word	Offset	Data Length	Contents
1	+0	N/A	Reserved for future IBM use.

Word	Offset	Data Length	Contents
2	+4	32 bytes	Address of the doubleword-aligned user area, which can be used by this CP exit routine as working storage.
3	+8	256 bytes	Address of the doubleword-aligned translate-and-test (TRT) table, which can be used by this CP exit routine as working storage.
4	+12	N/A	Address of the first CCW in the generated channel program. Note: The length of the channel program may vary. The last CCW that is part of the channel program will have the command chaining bit and the data chaining bit turned off.
5	+16	4096 bytes	Address of SEPPAG1 which contains separator page CCWs and data.
6	+20	4096 bytes	Address of SEPPAG2 which contains separator page data.
7	+24	SPFEND (bytes)	Address of the spool file control block (SPFBK) of the file being printed. Note: SPFEND is an equate in HCPSPFBK COPY.
8	+28	RDEVSZCK (doublewords)	Address of the real device control block (RDEV) of the printer the file is being printed on. Note: RDEVSZCK field is a one byte field in HCPRDEV COPY.
9	+32	RSPSIZE (doublewords)	Address of the real spool device block (RSPBK) of the printer the file is being printed on. Note: RSPSIZE is an equate in HCPRSPBK COPY.

Exit conditions

On return, CP Exit 4401 sets the standard register contents described in [“Standard Exit Conditions” on page 94](#).

Return codes

On return, R15 contains two return codes. The first half of the fullword contains the "CP exit control return code", which is explained in [“Standard Return Codes” on page 94](#). The second half of the fullword contains the "mainline return code", which is explained in the following table:

Return Code	Results
0	CP continues separator page processing.
4	CP does not perform the I/O to position the printer at the bottom of the form, but continues separator page processing.
8	CP terminates separator page processing for the current file.

Programming considerations

1. If the contents of SEPPAG1 are altered, its total length must not exceed 4096 bytes.
2. If the contents of SEPPAG2 are altered, its total length must not exceed 4096 bytes.
3. To bypass this step in separator page processing, no user exit code needs to be written. You can simply associate IBM-supplied entry point HCPSRC04 with this exit point.

4. To terminate separator page processing at this point, no user exit code needs to be written. You can simply associate IBM-supplied entry point HCPSRC08 with this exit point.

CP Exit 4402: Separator Page Perforation Printing or 3800 Positioning

Purpose

Use CP Exit 4402 to change the channel program which either:

- Prints lines of asterisks and dashes across the perforation of separator pages for impact printers, and positions the printer for printing of separator page data
- Positions 3800 printers for printing of separator page data

Point of processing

Process	CP Exit Attribute
Separator Page Perforation Printing or 3800 Positioning	<ul style="list-style-type: none">• Reentrant• MP• No locks held

CP Exit 4402 is called prior to printing of separator perforation lines (for impact printers) and positioning the printer to print separator page data (both impact printers and 3800s). The channel program is mapped by SEPPAG1 DSECT beginning at label SEPCWP1 in HCPSEPBK COPY.

Entry conditions

Register 1 points to a parameter list which is described below. For more information about other register contents, see [“Standard Entry Conditions”](#) on page 93.

Parameter list constant

The parameter list contains the following:

Word	Offset	Data Length	Contents
1	+0	N/A	Reserved for future IBM use.
2	+4	32 bytes	Address of a doubleword aligned User Area, which can be used by this CP exit routine as working storage.
3	+8	256 bytes	Address of a doubleword aligned translate-and-test (TRT) table, which can be used by this CP exit routine as working storage.
4	+12	N/A	Address of the first CCW in the generated channel program. Note: The length of the channel program may vary. The last CCW that is part of the channel program will have the command chaining bit and the data chaining bit turned off.
5	+16	4096 bytes	Address of SEPPAG1 which contains separator page CCWs and data.
6	+20	4096 bytes	Address of SEPPAG2 which contains separator page data.

Word	Offset	Data Length	Contents
7	+24	SPFEND (bytes)	Address of the spool file control block (SPFBK) of the file being printed. Note: SPFEND is an equate in HCPSPFBK COPY.
8	+28	RDEVSZCK (doublewords)	Address of the real device control block (RDEV) of the printer the file is being printed on. Note: RDEVSZCK field is a one byte field in HCPRDEV COPY.
9	+32	RSPSIZE (doublewords)	Address of the real spool device block (RSPBK) of the printer the file is being printed on. Note: RSPSIZE is an equate in HCPRSPBK COPY.

Exit conditions

On return, CP Exit 4402 sets the standard register contents described in [“Standard Exit Conditions” on page 94](#).

Return codes

On return, R15 contains two return codes. The first half of the fullword contains the "CP exit control return code", which is explained in [“Standard Return Codes” on page 94](#). The second half of the fullword contains the "mainline return code", which is explained in the following table:

Return Code	Results
0	CP continues separator page processing.
4	CP does not perform the I/O to print the perforation pattern (for impact printers) or position the printer for separator printing, but continues separator page processing.
8	CP terminates separator page processing for the current file.

Programming considerations

1. If the contents of SEPPAG1 are altered, its total length must not exceed 4096 bytes.
2. If the contents of SEPPAG2 are altered, its total length must not exceed 4096 bytes.
3. To bypass this step in separator page processing, no user exit code needs to be written. You can simply associate IBM-supplied entry point HCPSRC04 with this exit point.
4. To terminate separator page processing at this point, no user exit code needs to be written. You can simply associate IBM-supplied entry point HCPSRC08 with this exit point.

CP Exit 4403: Separator Page Printing

Purpose

Use CP Exit 4403 to alter the channel program which controls the printing of separator page data.

Point of processing

Process	CP Exit Attribute
Separator Page Printing	<ul style="list-style-type: none"> • Reentrant • MP • No locks held

CP Exit 4403 is called prior to printing of the first separator page. The channel program is in SEPPAG1 beginning at label SEPCWL1 in HCPSEPBK copy.

Entry conditions

Register 1 points to a parameter list which is described below. For more information about other register contents, see [“Standard Entry Conditions” on page 93.](#)

Parameter list constant

The parameter list contains the following:

Word	Offset	Data Length	Contents
1	+0	N/A	Reserved for future IBM use.
2	+4	32 bytes	Address of a doubleword-aligned User Area, which can be used by this CP exit routine as working storage.
3	+8	256 bytes	Address of a doubleword-aligned translate-and-test (TRT) table, which can be used by this CP exit routine as working storage.
4	+12	N/A	Address of the first CCW in the generated channel program. Note: The length of the channel program may vary. The last CCW that is part of the channel program will have the command chaining bit and the data chaining bit turned off.
5	+16	4096 bytes	Address of SEPPAG1 which contains separator page CCWs and data.
6	+20	4096 bytes	Address of SEPPAG2 which contains separator page data.
7	+24	SPFEND (bytes)	Address of the spool file control block (SPFBK) of the file being printed. Note: SPFEND is an equate in HCPSPFBK COPY.
8	+28	RDEVSZCK (doublewords)	Address of the real device control block (RDEV) of the printer the file is being printed on. Note: RDEVSZCK field is a one byte field in HCPRDEV COPY.
9	+32	RSPSIZE (doublewords)	Address of the real spool device block (RSPBK) of the printer the file is being printed on. Note: RSPSIZE is an equate in HCPRSPBK COPY.

Exit conditions

On return, CP Exit 4403 sets the standard register contents described in [“Standard Exit Conditions” on page 94.](#)

Return codes

On return, R15 contains two return codes. The first half of the fullword contains the "CP exit control return code", which is explained in ["Standard Return Codes" on page 94](#). The second half of the fullword contains the "mainline return code", which is explained in the following table:

Return Code	Results
0	CP continues separator page processing.
4	CP does not perform the I/O to print the first separator page, but continues separator page processing.
8	CP terminates separator page processing for the current file.

Programming considerations

1. If the contents of SEPPAG1 are altered, its total length must not exceed 4096 bytes.
2. If the contents of SEPPAG2 are altered, its total length must not exceed 4096 bytes.
3. To simply bypass this step in separator page processing, you can associate IBM-supplied entry point HCPSRC04 with this exit point.
4. To simply terminate separator page processing at this point, you can associate IBM-supplied entry point HCPSRC08 with this exit point.

CP Exit 4404: Second Separator Page Positioning

Purpose

Use CP Exit 4404 to change the channel program which positions the printer for printing of the second separator page. This channel program either positions impact printers at the bottom of the first separator page to repeat perforation printing, or positions 3800 printers to the top of the second separator page.

Point of processing

Process	CP Exit Attribute
Second Separator Page Positioning	<ul style="list-style-type: none">• Reentrant• MP• No locks held

CP Exit 4404 is called prior to positioning the printer to print the second separator page. The channel program is mapped by SEPPAG1 DSECT beginning at label SEPCWP1.

Entry conditions

Register 1 points to a parameter list which is described below. For more information about other register contents, see ["Standard Entry Conditions" on page 93](#).

Parameter list constant

The parameter list contains the following:

Word	Offset	Data Length	Contents
1	+0	N/A	Reserved for future IBM use.

Word	Offset	Data Length	Contents
2	+4	32 bytes	Address of a doubleword-aligned User Area, which can be used by this CP exit routine as working storage.
3	+8	256 bytes	Address of a doubleword-aligned translate-and-test (TRT) table, which can be used by this CP exit routine as working storage.
4	+12	N/A	Address of the first CCW in the generated channel program. Note: The length of the channel program may vary. The last CCW that is part of the channel program will have the command chaining bit and the data chaining bit turned off.
5	+16	4096 bytes	Address of SEPPAG1 which contains separator page CCWs and data.
6	+20	4096 bytes	Address of SEPPAG2 which contains separator page data.
7	+24	SPFEND (bytes)	Address of the spool file control block (SPFBK) of the file being printed. Note: SPFEND is an equate in HCPSFPBK COPY.
8	+28	RDEVSZCK (doublewords)	Address of the real device control block (RDEV) of the printer the file is being printed on. Note: RDEVSZCK field is a one byte field in HCPRDEV COPY.
9	+32	RSPSIZE (doublewords)	Address of the real spool device block (RSPBK) of the printer the file is being printed on. Note: RSPSIZE is an equate in HCPRSPBK COPY.

Exit conditions

On return, CP Exit 4404 sets the standard register contents described in [“Standard Exit Conditions” on page 94](#).

Return codes

On return, R15 contains two return codes. The first half of the fullword contains the "CP exit control return code", which is explained in [“Standard Return Codes” on page 94](#). The second half of the fullword contains the "mainline return code", which is explained in the following table:

Return Code	Results
0	CP continues separator page processing.
4	CP does not perform the I/O to position the printer for printing the second separator page, but continues separator page processing.
8	CP terminates separator page processing for the current file.

Programming considerations

1. If the contents of SEPPAG1 are altered, its total length must not exceed 4096 bytes.
2. If the contents of SEPPAG2 are altered, its total length must not exceed 4096 bytes.
3. To bypass this step in separator page processing, no user exit code needs to be written. You can simply associate IBM-supplied entry point HCPSRC04 with this exit point.

4. To terminate separator page processing at this point, no user exit code needs to be written. You can simply associate IBM-supplied entry point HCPSRC08 with this exit point.

CP Exit 4405: Second Separator Page Printing

Purpose

Use CP Exit 4405 to change the channel program which controls printing of the second separator page.

Point of processing

Process	CP Exit Attribute
Second Separator Page Printing	<ul style="list-style-type: none">• Reentrant• MP• No locks held

CP Exit 4405 is called prior to printing of the second separator page. The channel program is mapped by SEPPAG1 DSECT beginning at label SEPCWL1 in HCPSEPBK COPY.

Entry conditions

Register 1 points to a parameter list which is described below. For more information about other register contents, see [“Standard Entry Conditions” on page 93](#).

Parameter list constant

The parameter list contains the following:

Word	Offset	Data Length	Contents
1	+0	N/A	Reserved for future IBM use.
2	+4	32 bytes	Address of a doubleword-aligned User Area, which can be used by this CP exit routine as working storage.
3	+8	256 bytes	Address of doubleword-aligned translate-and-test (TRT) table, which can be used by this CP exit routine as working storage.
4	+12	N/A	Address of the first CCW in the generated channel program. Note: The length of the channel program may vary. The last CCW that is part of the channel program will have the command chaining bit and the data chaining bit turned off.
5	+16	4096 bytes	Address of SEPPAG1 which contains separator page CCWs and data.
6	+20	4096 bytes	Address of SEPPAG2 which contains separator page data.
7	+24	SPFEND (bytes)	Address of the spool file control block (SPFBK) of the file being printed. Note: SPFEND is an equate in HCPSPFBK COPY.

Word	Offset	Data Length	Contents
8	+28	RDEVSZCK (doublewords)	Address of the real device control block (RDEV) of the printer the file is being printed on. Note: RDEVSZCK field is a one byte field in HCP RDEV COPY.
9	+32	RSPSIZE (doublewords)	Address of the real spool device block (RSPBK) of the printer the file is being printed on. Note: RSPSIZE is an equate in HCP RSPBK COPY.

Exit conditions

On return, CP Exit 4405 sets the standard register contents described in [“Standard Exit Conditions” on page 94](#).

Return codes

On return, R15 contains two return codes. The first half of the fullword contains the "CP exit control return code", which is explained in [“Standard Return Codes” on page 94](#). The second half of the fullword contains the "mainline return code", which is explained in the following table:

Return Code	Results
0	CP continues separator page processing.
4	CP does not perform the I/O to print the second separator page, but continues separator page processing.
8	CP terminates separator page processing for the current file.

Programming considerations

1. If the contents of SEPPAG1 are altered, its total length must not exceed 4096 bytes.
2. If the contents of SEPPAG2 are altered, its total length must not exceed 4096 bytes.
3. To bypass this step in separator page processing, no user exit code needs to be written. You can simply associate IBM-supplied entry point HCPSRC04 with this exit point.
4. To terminate separator page processing at this point, no user exit code needs to be written. You can simply associate IBM-supplied entry point HCPSRC08 with this exit point.

CP Exit 4406: Separator Page Post-Print Positioning

Purpose

Use CP Exit 4406 to change the channel program which positions the printer to begin printing spool file data after the separator pages have been printed.

Point of processing

Process	CP Exit Attribute
Separator Page Post-Print Positioning	<ul style="list-style-type: none"> • Reentrant • MP • No locks held

CP Exit 4406 is called prior to positioning the printer to begin printing the data in the spool file. The channel program is mapped by SEPPAG1 DSECT beginning at label SEPCCW1 in HCPSEPBK COPY.

Entry conditions

Register 1 points to a parameter list which is described below. For more information about other register contents, see [“Standard Entry Conditions”](#) on page 93.

Parameter list constant

The parameter list contains the following:

Word	Offset	Data Length	Contents
1	+0	N/A	Reserved for future IBM use.
2	+4	32 bytes	Address of a doubleword-aligned User Area, which can be used by this CP exit routine as working storage.
3	+8	256 bytes	Address of a doubleword-aligned translate-and-test (TRT) table, which can be used by this CP exit routine as working storage.
4	+12	N/A	Address of the first CCW in the generated channel program. Note: The length of the channel program may vary. The last CCW that is part of the channel program will have the command chaining bit and the data chaining bit turned off.
5	+16	4096 bytes	Address of SEPPAG1 which contains separator page CCWs and data.
6	+20	4096 bytes	Address of SEPPAG2 which contains separator page data.
7	+24	SPFEND (bytes)	Address of the spool file control block (SPFBK) of the file being printed. Note: SPFEND is an equate in HCPSPFBK COPY.
8	+28	RDEVSZCK (doublewords)	Address of the real device control block (RDEV) of the printer the file is being printed on. Note: RDEVSZCK field is a one byte field in HCPRDEV COPY.
9	+32	RSPSIZE (doublewords)	Address of the real spool device block (RSPBK) of the printer the file is being printed on. Note: RSPSIZE is an equate in HCPRSPBK COPY.

Exit conditions

On return, CP Exit 4406 sets the standard register contents described in [“Standard Exit Conditions”](#) on page 94.

Return codes

On return, R15 contains two return codes. The first half of the fullword contains the "CP exit control return code", which is explained in ["Standard Return Codes" on page 94](#). The second half of the fullword contains the "mainline return code", which is explained in the following table:

Return Code	Results
0	CP continues separator page processing.
4	CP does not perform the I/O to position the printer to begin printing spool file data, but continues separator page processing.
8	CP terminates separator page processing for the current file.

Programming considerations

1. If the contents of SEPPAG1 are altered, its total length must not exceed 4096 bytes.
2. If the contents of SEPPAG2 are altered, its total length must not exceed 4096 bytes.
3. To bypass this step in separator page processing, no user exit code needs to be written. You can simply associate IBM-supplied entry point HCPSRC04 with this exit point.
4. To terminate separator page processing at this point, no user exit code needs to be written. You can simply associate IBM-supplied entry point HCPSRC08 with this exit point.

CP Exit 4407: Trailer Page Processing

Purpose

Use CP Exit 4407 to change or replace the printed data on the separator trailer page, and/or to modify the channel program which controls the printing of the separator trailer page.

Point of processing

Process	CP Exit Attribute
Trailer Page Processing	<ul style="list-style-type: none">• Reentrant• MP• No locks held

CP Exit 4407 is called during separator trailer page processing, prior to printing the separator trailer page. The data that will be printed on the separator page and the channel program which controls printing of the separator trailer page have been created at this point in the process. The separator trailer page data is mapped by SEPTRL DSECT in HCPSEPBK COPY. The channel program is mapped by SEPTRL DSECT beginning at label SEPTCCW1.

Entry conditions

Register 1 points to a parameter list which is described below. For more information about other register contents, see ["Standard Entry Conditions" on page 93](#).

Parameter list constant

The parameter list contains the following:

Word	Offset	Data Length	Contents
1	+0	N/A	Reserved for future IBM use.
2	+4	32 bytes	Address of a doubleword-aligned User Area, which can be used by this CP exit routine as working storage.
3	+8	256 bytes	Address of a doubleword-aligned translate-and-test (TRT) table, which can be used by this CP exit routine as working storage.
4	+12	N/A	Address of the first CCW in the generated channel program. Note: The length of the channel program may vary. The last CCW that is part of the channel program will have the command chaining bit and the data chaining bit turned off.
5	+16	4096 bytes	Address of SEPTRL which contains separator trailer page CCWs and data.
6	+20	SPFEND (bytes)	Address of the spool file control block (SPFBK) of the file being printed. Note: SPFEND is an equate in HCPSFBK COPY.
7	+24	RDEVSZCK (doublewords)	Address of the real device control block (RDEV) of the printer the file is being printed on. Note: RDEVSZCK field is a one byte field in HCPRDEV COPY.
8	+28	RSPSIZE (doublewords)	Address of the real spool device block (RSPBK) of the printer the file is being printed on. Note: RSPSIZE is an equate in HCPRSPBK COPY.

Exit conditions

On return, CP Exit 4407 sets the standard register contents described in [“Standard Exit Conditions” on page 94](#).

Return codes

On return, R15 contains two return codes. The first half of the fullword contains the "CP exit control return code", which is explained in [“Standard Return Codes” on page 94](#). The second half of the fullword contains the "mainline return code", which is explained in the following table:

Return Code	Results
0	CP prints the separator trailer page.
4	CP does not print the separator trailer page.
8	CP does not print the separator trailer page.

Programming considerations

1. If the contents of SEPTRL DSECT are altered, its total length must not exceed 4096 bytes.
2. To terminate separator trailer page processing at this point, no user exit code needs to be written. You can simply associate IBM-supplied entry point HCPSRC04 or HCPSRC08 with this exit point.

CP Exit Point and Module Reference

The following table provides a reference to the IBM-defined CP exit points and the CP modules from which they are called. This information is provided for diagnostic purposes only.

CP Exit	Name	Module
00E0	Startup Pre-Prompt Processing	HCPISX
00E1	Startup Post-Prompt Processing	HCPISX
0FFB	Post-Authorization Command Processing	HCPCMD
1100	LOGON Command Pre-Parse Processing	HCPLGX
1101	Logon Post-Parse Processing	HCPLGX
1110	VMDBK Pre-Logon Processing	HCPLGX
117F	Logon Final Screening	HCPLGX
11C0	Logoff Initial Screening	HCPUSP
11FF	Logoff Final Screening	HCPUSP
1200	DIAL Command Initial Screening	HCPDIA
1201	DIAL Command Final Screening	HCPDIA
1210	Messaging Commands Screening	HCPMFS
3FE8	SHUTDOWN Command Screening	HCPSHS
4400	Separator Page Data Customization	HCPSEP
4401	Separator Page Pre-Perforation Positioning	HCPSEP
4402	Separator Page Perforation Printing or 3800 Positioning	HCPSEP
4403	Separator Page Printing	HCPSEP
4404	Second Separator Page Positioning	HCPSEP
4405	Second Separator Page Printing	HCPSEP
4406	Separator Page Post-Print Positioning	HCPSEP
4407	Trailer Page Processing	HCPSEQ

Appendix B. Dynamic Exit Points

The conditions under which exit routines associated with dynamic exit points are invoked, and the conventions of such invocations, are determined largely by the way that the exit points are defined. However, there are some conventions that apply regardless of where the exit point is placed.

Point of Processing

The point of processing and the associated attributes (for example, the locks held) are determined by the placement of the dynamic exit.

Entry Conditions

Register 1 points to a parameter list whose format is determined in part by how the exit point is defined. The invariant portion of the parameter list (words 1 through 3) is described below, along with a variant portion (beginning with word 4) that includes zero or more user-defined parameters. The contents of other registers are described in [“Standard Entry Conditions”](#) on page 93.

Parameter List Contents

Word	Offset	Data Length	Contents
1	+0	128 bytes	Address of the save area (SAVBK) containing the registers at the point where the exit was taken and from which the registers will be restored upon completion of exit processing.
2	+4	32 bytes	Address of the doubleword-aligned User Area, which can be used by this CP exit routine as working storage.
3	+8	256 bytes	Address of the doubleword-aligned translate-and-test (TRT) table, which can be used by this CP exit routine as working storage.
4	+12	Context- dependent	User-defined parameter 1
5	+16	Context- dependent	User-defined parameter 2
:			

Exit Conditions

On return, a dynamic exit restores the original contents of registers 0 through 15. However, if the exit routine chooses to modify the register values in the SAVBK pointed to by the first word of the exit parameter list, it can affect the values restored to the corresponding registers.

Return Codes

On return, R15 contains two return codes. The first half of the fullword contains the "CP exit control return code". These return codes are explained in the table under [“Standard Return Codes”](#) on page 94. The second half of the fullword contains the "mainline return code", which determines whether or not the

Dynamic Exit Points

instruction at the exit point is executed. Return code 0 tells CP to execute the instruction; any other value tells CP to skip the instruction.

Appendix C. CPXLOAD Directives

This appendix describes the CPXLOAD directives that control the dynamic loading of CP routines into the system execution space using the CPXLOAD command or configuration file statement.

The table below lists and briefly explains the CPXLOAD directives that are available to you. These CPXLOAD directives can be used in TEXT files, in control files, or in members of TXTLIBs.

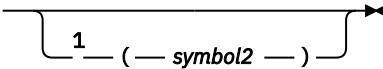
Table 6. CPXLOAD Directives

CPXLOAD Directive	Description	Page
CHANGE	Temporarily renames or deletes an external symbol associated with CP routines dynamically loaded using a CPXLOAD command or configuration file statement. This lets you make temporary changes during the loading of a TEXT file without requiring you to reassemble the file.	“CHANGE Directive” on page 152
EXPAND	Increases the size of the control section (CSECT) when dynamically loading CP routines using a CPXLOAD command or configuration file statement.	“EXPAND Directive” on page 154
INCLUDE	Reads and processes another file when dynamically loading CP routines using a CPXLOAD command or configuration file statement.	“INCLUDE Directive” on page 156
OPTIONS	Sets the defaults for dynamically loading CP routines into the system execution space using a CPXLOAD command or configuration file statement.	“OPTIONS Directive” on page 158

For information about how to read syntax diagrams, see [“Syntax, Message, and Response Conventions” on page xiii](#).

CHANGE Directive

Format

➡ CHANGE — *symbol1* — 

Notes:

¹ If you specify *symbol2*, there must not be any spaces between *symbol1* and the left parenthesis delimiter of *symbol2*.

Purpose

Use the CHANGE directive to temporarily rename or delete an external symbol associated with CP routines dynamically loaded using a CPXLOAD command or configuration file statement. This lets you make temporary changes during the loading of a TEXT file without requiring you to reassemble the file.

How to specify

CHANGE directives must start after column 1 of the input record and must be specified immediately before the first external symbol (*symbol1*) is defined. This directive remains in effect until it encounters an END record in the TEXT file.

For example, suppose you had an INCLUDE directive which read and processed another file. In that file, you defined an external symbol that you wanted to rename or delete. You would place the CHANGE directive immediately before the INCLUDE directive.

Operands

symbol1

is the external symbol that you are renaming or deleting. The variable *symbol1* must be a 1- to 8-character alphanumeric string.

(*symbol2*)

is the new external symbol name that you want CP to use temporarily during loading instead of *symbol1*. The variable *symbol2* must be a 1- to 8-character alphanumeric string.

If you omit *symbol2*, you are telling CP to temporarily delete *symbol1* during loading.

Usage notes

1. Someday, you may need to load a file that duplicates the entry point names already known to CP. Rather than changing and assembling the source file, you can use the CHANGE directive. This saves you from assembling again and helps avoid potential errors.
2. If you do not specify *symbol2* on a CHANGE directive, CP temporarily deletes *symbol1*. If *symbol1* is a control section (CSECT), CP will delete that CSECT and all of its internal entry point names. That is, CP treats the CSECT as if it were not in the input file.

If *symbol1* is a simple external symbol within a CSECT, CP ignores that external symbol, but retains all other parts of the CSECT. That is, CP treats the external symbol as if it had not been declared in the CSECT. An external reference to that external symbol is treated as an unresolved reference.

Examples

1. To have CP temporarily change the references from HCPCVTHB to HCPCVTDB in the file being loaded, use the following:

```
change  hcpcvthb(hcpcvtddb)
```

Messages

HCP6704E Missing token at end of line

HCP6706E Invalid entry point name - *symbol*

EXPAND Directive

Format

➤ EXPAND — *csect* — (— *nnnn* —) ➤

Purpose

Use the EXPAND directive to increase the size of the control section (CSECT) when dynamically loading CP routines using a CPXLOAD command or configuration file statement.

How to specify

EXPAND directives must start after column 1 of the input record and must be placed before the specified control section (*csect*) is defined.

Operands

csect

is the name of the CSECT in the file to be loaded whose size you want to increase. The variable *csect* must be a 1- to 8-character alphanumeric string.

(*nnnn*)

tells CP how many more bytes to add to the current CSECT size. The variable *nnnn* is a decimal number between 1 and 4096.

Usage notes

1. When CP expands the size of the CSECT, it adds the expansion area at the end of the CSECT.
2. You may get a larger expansion area than you asked for, because CP always rounds the expansion area size up to a double-word boundary.
3. Patching is the most likely use you will have for the expansion area.
4. When using the EXPAND directive, be careful not to increase the size of your program beyond its own design limitations. For example, if space is added to a CSECT beyond the range of its base registers, then the program will not have addressability to that area.
5. When using the EXPAND and CHANGE directives together, be careful to expand the correct CSECT name. That is, if you use the CHANGE directive to change the name of the CSECT, then the EXPAND directive must refer to the new CSECT name.

Examples

1. To have CP increase the size of a CSECT named QQQCSECT by 16 bytes, use the following:

```
expand   qqqcsect(16)
```

2. To change the CSECT name to XXXCSECT and then increase its size by 16 bytes, use the following:

```
change   qqqcsect(xxxcsect)
expand   xxxcsect(16)
```

Messages

HCP002E Invalid operand - *operand*

HCP6706E Invalid entry point name - *csect*

INCLUDE Directive

Format

➤ INCLUDE — *fn* — *ft* — MEMber — *member* ➤

Purpose

Use the INCLUDE directive to read and process another file when dynamically loading CP routines using a CPXLOAD command or configuration file statement.

How to specify

INCLUDE directives must start after column 1 of the input record and can be placed anywhere in the dynamically loaded routine. For more information about how CP processes INCLUDE directives, see Usage Note “2” on page 156.

Operands

fn

is the file name of the CMS file that you want to include.

ft

is the file type of the CMS file that you want to include.

MEMBER *member*

is the name of the member in the TXTLIB that you want included.

You can specify an asterisk (*) to include all the base members in the partitioned data set.

You can use generic member names to request a specific subset of files. A generic member name is a 1- to 8-character string with asterisks (*) in place of 1 or more characters and percent signs (%) in place of exactly 1 character. For example:

```
... member hc%p*
```

includes all members that start with HC and have P as their fourth character.

Note: You can only specify MEMBER when the file is actually a CMS partitioned data set (PDS).

Usage notes

1. The file that you are loading and the file that you are including must both reside on the same minidisk.
2. The placement of the INCLUDE directive in the file that you are loading is important. CP processes the file that you are loading until it finds an INCLUDE. Then, CP processes the included file until it reaches the end of the file (or until it reaches an LDT record in a file containing TEXT records). After processing the included file, CP returns to processing the file that you are loading at the line just after the INCLUDE.
3. Members of a TXTLIB are either base members or alias members. A base member and its alias members all start at the same record in the file. A base member is the first member in the file's directory that starts at a different record than all previous directory entries. All subsequent members in the directory that start at the same record are alias members to the base member. To display the list of base member names and their alias member names, use the CPLISTFILE command. For more information about CPLISTFILE, see [z/VM: CP Commands and Utilities Reference](#).
4. CP will reject processing an INCLUDE directive if the included file would cause an INCLUDE loop. For example, if a file includes another file which in turn includes the first file, CP would process the first

file, but not the second file, because the two files would just loop back and forth including each other and you would never finish loading your routines.

Examples

1. To have CP read and process another file when dynamically loading CP routines, place the following in the file being loaded:

```
include fn1 text
```

2. To have CP read and process all members of a TXTLIB, place the following in the file being loaded:

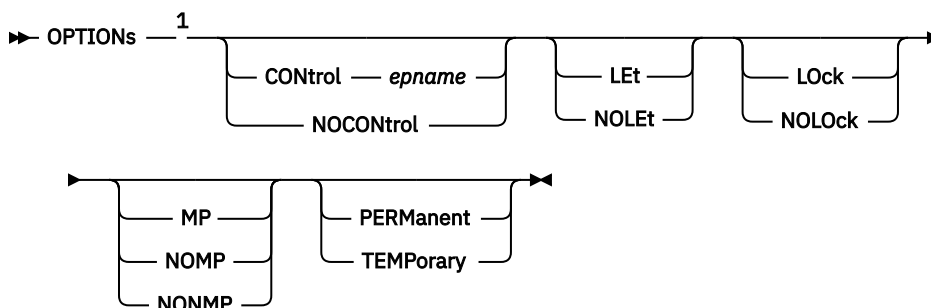
```
include subrtns txtlib member *
```

Messages

- HCP003E Invalid option - statement contains extra option(s) starting with *option*
- HCP6703E File *fn ft fm* not found.
- HCP6704E Missing token at end of line
- HCP6706E The variations of this message are as follows:
 - Invalid file name - *fn*
 - Invalid file type - *ft*
 - Invalid file member name - *member*
- HCP6710E Including file *fn ft* would cause an include loop -- statement ignored.
- HCP8019E Invalid placement:
- HCP8040E File *fn ft fm* Record *nnn*
Copy_of_record

OPTIONS Directive

Format



Notes:

¹ You must specify at least one operand. If you specify more than one operand, you can specify them in any order.

Purpose

Use the **OPTIONS** directive to set the defaults for dynamically loading CP routines into the system execution space using a **CPIXLOAD** command or configuration file statement.

How to specify

OPTIONS directives must start after column 1 of the input record. You can place them anywhere between the control sections (CSECTs) of the dynamically loaded CP routine and you can include as many directives as needed. The attributes that you specify on the OPTIONS directive (for example, NOMP) will be set for all external symbols in the CSECT. Within a CSECT, if you need to specify different OPTIONS attributes for different external symbols, you must split the CSECT into multiple CSECTs.

The defaults that you set on an `OPTIONS` directive remain in effect only for the duration of the current CPXLOAD operation. After all the routines for that CPXLOAD operation are loaded, the defaults revert back to those defined by IBM.

If you specify more than one **OPTIONS** directive that specify conflicting defaults, the latest default specification overrides any previous default specifications, with the following exceptions: **CONTROL**, **NOCONTROL**, **PERMANENT**, and **TEMPORARY**. Once specified, you cannot override the defaults for these 4 operands in a subsequent **OPTIONS** directive. However, you can override these defaults using a **CPXLOAD** command or configuration file statement. Defaults specified on **CPXLOAD** always override defaults specified on **OPTIONS** directives.

Operands

CONtrol *epname*

tells CP to call the specified entry point after dynamically loading the CP routines and before processing a CPXUNLOAD request. You can load the routines containing the specified entry point either before or within this CPXLOAD request. The variable *epname* must be a 1- to 8-character string. The first character must be alphabetic or one of the following special characters: dollar sign (\$), number sign (#), underscore (_), or at sign (@). The rest of the string can be alphanumeric characters, the 4 special characters (\$, #, _, and @), or any combination thereof.

Note: Normally, if CP cannot find an entry point when processing a routine, it ignores the unknown entry point and continues normal processing. This is not true when you specify the CONTROL *epname* operand. If CP cannot find the entry point you specify on CONTROL, CP terminates processing your CPXLOAD request and will not load the routines into its storage.

NOCONTROL

tells CP not to call an entry point after loading CP routines and before processing a CPXUNLOAD request.

LEt

tells CP to ignore any records that are completely blank or that contain an unexpected value in column 1, and to continue the load operation. This accommodates the non-commented information that can be left in a TEXT file by an assembler utility (such as VMFHASM or VMFHLASM).

NOLEt

tells CP to stop the load operation when an unexpected value is encountered in column 1. Column 1 is expected to contain an asterisk (*) to denote a comment, X'02' for a TEXT record, or a blank for a possible CPXLOAD directive.

LOck

is provided for compatibility purposes only and has no effect.

NOLOck

is provided for compatibility purposes only and has no effect.

MP

is provided for compatibility purposes only and has no effect.

NOMP**NONMP**

is provided for compatibility purposes only and has no effect.

PERManent

tells CP that the dynamically loaded CP routines are to remain a part of CP until a CP SHUTDOWN command is issued or a software-initiated restart (bounce) occurs. This means you cannot use the CPXUNLOAD command to remove these routines.

TEMPorary

tells CP that the dynamically loaded CP routines can be unloaded in the future with a CPXUNLOAD command.

Usage notes

1. To dynamically load the CP routines into the system execution space, use the CPXLOAD command or configuration file statement. For more information on the CPXLOAD command, see *z/VM: CP Commands and Utilities Reference*. For more information on the CPXLOAD configuration file statement, see *z/VM: CP Planning and Administration*.
2. The purpose of the OPTIONS directive is to allow you to specify your CPXLOAD options before issuing the actual command or configuration file statement. However, you can specify options on both the OPTIONS directive and the CPXLOAD command or statement. If you do so and any of the options conflict, CP uses the options from the CPXLOAD command or statement. For example, suppose you specify PERMANENT on the OPTIONS directive and TEMPORARY on the CPXLOAD command, CP will use TEMPORARY.

Examples

1. To have CP set the default for dynamically loading CP routines that entry points are not multiprocessor capable and must be dispatched on the master processor, place the following in the file being loaded:

```
options nomp
```

Messages

HCP002E	Invalid operand - <i>operand</i>	HCP2768E	Missing {file name file type number entry point name file member name}
HCP2757E	This option[, or its opposite,] has already been specified - <i>option</i>		

HCP6704E	Missing token at end of line
HCP8019E	Invalid placement:

HCP8040E	File <i>fn ft fm</i> Record <i>nnn</i> <i>Copy_of_record</i>
-----------------	---

Appendix D. CP Files of Interest

This appendix lists and briefly describes the CP control blocks, macros, and modules that you can use with your dynamically loaded CP routines.

CP Control Blocks and Macros

The CP component of z/VM has many data areas, control blocks, and macros that govern such things as real I/O, virtual I/O, system initialization, dispatching, and so forth. The intent of this section is to focus the reader on a few of the control blocks and macros that are related to virtual machine characteristics, real and virtual I/O control blocks, system operation, and system-equated values and valid device types. This list of control blocks and macros is neither complete nor exhaustive. It is supplied here only to inform you that these control blocks and macros exist. It does not imply that you should or must use them.

Control Block	Purpose
ADDDL	(add double length binary) generates the necessary code to add 2 fixed-point doubleword numbers.
HPCALL	(VM general purpose macro to call a module) generates the necessary code to call a CP entry point.
HPCASRC	(case of return code) generates the necessary code to construct a branch table for a return code passed in register 15.
HPCMPBK	(component ID block) contains fields for customer use. Each component ID can be assigned a separate CMPBK.
HPCMPID	(define component ids for macro processing) sets global macro variables so that other macros may use component ids other than HCP for their processing.
HPCONSL	(console interface macro) generates the necessary code to write a message or response, or to read input from the terminal.
HCPDROP	(structured drop statement) generates the necessary code to assist the HCPUSING macro in identifying conflicts between USING and DROP. A conflict arises whenever an area is addressed by more than one register or a single register is used to provide addressability to more than one area.
HCPDVTYPE	(constants for device type information) contains constants for all of the defined device classes, types, models, and device-specific feature information.
HCPENTER	(enter a module) generates the necessary code for most CP module entry points to save registers and provide addressability.
HCPEPILG	(epilogue macro) generates the necessary code to produce an epilogue containing the unique characters (the last 2) of every valid entry definition and entry point within that module, along with the offset to that entry point definition. This allows diagnostic and dump modules to locate every CP entry point. This macro also generates the "END" statement for the assembler.
HCPEQUAT	(equate symbols) contains all of the standard system-equated values used by CP modules. These include equated values such as: hardware architecture definition bits, scheduler values, system name table values, terminal values, and values for general and floating point registers.
HCPEQXIT	(equates for exit control) contains equated values for CP exit numbers.

Control Block	Purpose
HCPEXIT	(exit linkage) generates the necessary code to exit and return to the caller from any routine entered using the HCPCALL macro. It provides the ability to reload registers from the same save area used at the entry point established by HCPENTER.
HCP EXTRN	(generate an EXTRN for a symbol) generates the necessary code to generate an EXTRN for a symbol, if one has not been previously generated.
HCPGETST	(get storage macro) generates the necessary code to get a block of free storage, or to get one or more contiguous 4 KB pages.
HCPLCALL	(local call linkage macro) generates the necessary code to generate a call with dynamic save area linkage to a function that is local (that is, in the same module).
HCPLENTN	(local function entry linkage) generates the necessary code for a local entry point to save registers and provide addressability.
HCPLEXIT	(return from local function) generates the necessary code to generate linkage to return to the caller of a local function.
HCPPROLG	(prologue for module) generates the necessary code to establish attributes for the entire module.
HCPRDEV	(real device control block) describes a real I/O device. Each real I/O device known to CP is represented by an HCPRDEV control block. The RDEV contains information about the type and class of a specific device, the status of the device as well as queue anchors for I/O requests. The RDEV also contains pointers to other I/O control blocks associated with a real I/O device.
HCPRELST	(release storage macro) generates the necessary code to release a block of free storage, or to release one or more contiguous 4 KB pages.
HCPSYSCM	<p>(system common area) contains system-wide variables, counters, pointers, and constants. Some of these include:</p> <ul style="list-style-type: none"> • The volume serial ID of the system residence, warm start, and checkpoint volumes • Pointers to real device control blocks • Spool file block pointers • System counters for logged on users • Paging slots available • Reserved pages • Paging load. <p>The system common area also contains constants such as the size of real storage, directory control information, and the system residence volume characteristics and definition information.</p>
HCPUSING	(USING assembler statement) generates the necessary code to provide the USING assembler statement with the ability to identify potential conflicts between USING and DROP. A conflict arises whenever an area is addressed by more than one register or a single register is used to provide addressability to more than one area.
HCPVDEV	(virtual device control block) describes the status of a virtual device accessible by a virtual machine. As with the RDEV block, the VDEV block also contains information about the type and class of a specific virtual device, the status of that device, and information about the last I/O that was active.

Control Block	Purpose
HCPVMDBK	(virtual machine definition block) describes the characteristics of a virtual machine. This includes its user logon identification, accounting information, virtual registers, dispatching priority, and machine option settings. Every virtual machine on the system is represented by a HCPVMDBK control block.
HCPXCRBK	(CP exit call request block) contains data fields used by CP to control the calling of CP exit routines. At entry, each CP exit routine is passed the address of its XCRBK in register 2.
HCPXITPL	(exit parameter list) describes parameter lists for each CP exit point.
HCPXREF	(force a cross reference) generates the necessary code to force symbols to show up in the cross reference when they are not referenced explicitly by name.
IPARML	(IUCV/APPC parameter list and external interrupt mapping DSECT) maps the parameter list used when an IUCV or APPC/VM function is issued. Also, IPARML maps the external interrupt buffer when an IUCV or APPC/VM external interrupt is reflected to a virtual machine or CP system service.
SUBDL	(subtract double length binary) generates the necessary code to subtract a fixed-point doubleword number from another fixed-point doubleword number.
VMCPARM	(VMCF communications parameter list) specifies the virtual machine communication facility (VMCF) function to be executed along with other information required by VMCF.
VRDCBLOK	(virtual/real device characteristics block) describes a specific virtual device. This description includes both virtual and real device information as well as virtualized read device characteristics data.

For a list of macros and copyfiles which are provided as programming interfaces, see *z/VM: CP Programming Services*. For a complete description of all of the CP control blocks, see the IBM VM operating system home page. For more information about the macros, see the documentation in the macros.

CP Modules

The CP component of z/VM has many different modules that control system initialization, spooling, real I/O, virtual I/O, system services, dispatching, scheduling, and so forth. This section is being provided to give the reader examples of some of the most commonly called CP routines and their function.

CP Routine	Purpose
HCPBITSF	Turns off a bit in a bit map.
HCPBITSN	Turns on a bit in a bit map.
HCPCVTRM	Convert from binary to decimal digits and trim off any leading zeros.
HPCVTHB	Convert from hexadecimal displayable digits to binary.
HCPIOLAR	Acquire an RDEV lock.
HCPIOLRR	Release an RDEV lock.
HCPSCNAR	Add an RDEV to the radix tree.
HCPSCNRU	Find an RDEV in the radix tree.
HCPZIACC	Access a CP-accessed disk.

CP Routine	Purpose
HCPZICLS	Close a file on a CP-accessed disk.
HCPZIGET	Read a record from a file on a CP-accessed disk.
HCPZIOPN	Open a file on a CP-accessed disk.
HCPZIPUT	Write a record to a file.
HCPZISTA	Provide information about the existence of a file on a CP-accessed disk (STATE).
HCPZPRPC	Parse a partial command contained in the general system data block (GSDBK).
HCPZPRPG	Parse a general system data block (GSDBK).

For more information on these CP routines and the services they provide, see the individual CP module.

Note: It is possible that the names of these routines could change. For example, a routine may be moved from one module to another as a result of a module split. We recommend that you confirm the existence of any routines that you use when you apply service.

Appendix E. CP Accounting Exit (Module HCPACU)

You can make system-wide changes to the accounting records that CP collects by adding source code to module HCPACU. HCPACU is an installation-wide exit.

HCPACU supports installation-supplied code for examining and modifying CP accounting records. HCPACU is included in the CP nucleus at system generation. Later, during system operation, the CP accounting function calls HCPACU whenever any of the following occurs:

- A virtual machine is started or stopped
- Accounting commands are entered by an operator or a virtual machine
- CP does checkpointing operations.

However, if you do not add any code to HCPACU, the module performs no meaningful function.

HCPACU contains two entry points, HCPACUON and HCPACUOF. [Figure 10 on page 165](#) provides an overview of the IBM-supplied HCPACU module, showing the entry points and the areas where you can add your own source code.

```

HCPACU    COPY  HCOPTNS
          HCPPROLG ATTR=(RESIDENT),BASE=(R12)
*
          COPY  AOFPARM           parameter list for HCPACUOF
          COPY  AONPARM           parameter list for HCPACUON
          COPY  HCPEQUAT          common system equates
          COPY  HCPPFXPG          prefix page
          COPY  HCPSAVBK          savearea block
*
          HCPUSING PFXPG,0
*
HCPACUON  HCPENTER CALL,SAVE=DYNAMIC
          HCPUSING AONPARM,R1
*****
*
*   Installation-supplied code can be inserted here.
*
*****
          HCPDROP R1
          SLR   R15,R15           Set return code of 0
ACUONEXT  DS    0H
          ST    R15,SAVER15       Store return code
          L     R11,SAVER11       Restore register 11
          HCPEXIT EP=HCPACUON
          HCPDROP R13
*
HCPACUOF  HCPENTER CALL,SAVE=PFXBALSV
          HCPUSING AOFPARM,R1
          LA    R13,PFXBALSV     Establish savearea address
          HCPUSING SAVBK,R13
*****
*
*   Installation-supplied code can be inserted here.
*
*****
          HCPDROP R1
          SLR   R15,R15           Set return code of 0
ACUOFEXT  DS    0H
          ST    R15,SAVER15       Store return code
          L     R11,SAVER11       HCPEXIT does actual load
*
          HCPEXIT EP=HCPACUOF
          HCPDROP R0,R13
*
          HCPEPILG
```

Figure 10. Module HCPACU Overview

HCPACU has the following requirements and restrictions:

1. HCPACU is written in Assembler H coding language.
2. HCPPROLG, HCPUSING, HCPENTER, HCPDROP, HCPEXIT, and HCPEPILG are IBM-supplied macros (in HCPPSI MACLIB) that must be preserved in HCPACU and used only as shown.
3. The attributes of HCPACU must be REENTRANT and RESIDENT, because HCPACU is called during CP initialization and during CP checkpointing operations.
4. Links and accesses from HCPACU to other modules, data areas, and control blocks are not supported.
5. IBM reserves the right to modify the code associated with this exit. IBM will attempt to make any changes as transparent as possible to the customer.
6. Upon entry to both HCPACUON and HCPACUOF, register 13 points to a save area mapped by HCPSAVBK, shown in [Figure 11 on page 166](#). The caller's registers are saved in the save area. A ten-fullword work area, beginning at label SAVEWRK, is available for use by installation-supplied code.
7. Register 11, which points to the address of the dispatched VMDBK, and Register 13, which points to the address of the save area, must not be modified by installation-supplied code.

SAVBK	DSECT		
	DS	6F	Reserved for IBM use
SAVEREGS	DS	0XL64	CALLER'S REGISTERS - R0 to R15
SAVER0	DS	F	CALLER'S SAVED REGISTER 0
SAVER1	DS	F	CALLER'S SAVED REGISTER 1
SAVER2	DS	F	CALLER'S SAVED REGISTER 2
SAVER3	DS	F	CALLER'S SAVED REGISTER 3
SAVER4	DS	F	CALLER'S SAVED REGISTER 4
SAVER5	DS	F	CALLER'S SAVED REGISTER 5
SAVER6	DS	F	CALLER'S SAVED REGISTER 6
SAVER7	DS	F	CALLER'S SAVED REGISTER 7
SAVER8	DS	F	CALLER'S SAVED REGISTER 8
SAVER9	DS	F	CALLER'S SAVED REGISTER 9
SAVER10	DS	F	CALLER'S SAVED REGISTER 10
SAVER11	DS	F	CALLER'S SAVED REGISTER 11
SAVER12	DS	F	CALLER'S SAVED REGISTER 12
SAVER13	DS	F	CALLER'S SAVED REGISTER 13
SAVER14	DS	F	CALLER'S SAVED REGISTER 14
SAVER15	DS	F	CALLER'S SAVED REGISTER 15
SAVEWRK	DS	0XL40	WORKAREA FOR CALLEE
SAVEWRK0	DS	F	WORKAREA FOR CALLEE
SAVEWRK1	DS	F	WORKAREA FOR CALLEE
SAVEWRK2	DS	F	WORKAREA FOR CALLEE
SAVEWRK3	DS	F	WORKAREA FOR CALLEE
SAVEWRK4	DS	F	WORKAREA FOR CALLEE
SAVEWRK5	DS	F	WORKAREA FOR CALLEE
SAVEWRK6	DS	F	WORKAREA FOR CALLEE
SAVEWRK7	DS	F	WORKAREA FOR CALLEE
SAVEWRK8	DS	F	WORKAREA FOR CALLEE
SAVEWRK9	DS	F	WORKAREA FOR CALLEE

Figure 11. HCPSAVBK Save Area

Entry Point HCPACUON

Entry point HCPACUON supports installation-supplied code for the IBM 3277 Operator Identification Card Reader feature. You can add code to examine or validate the data read from an identification card and to accept or reject operator connection to the system.

HCPACUON is called when a LOGON command is entered from a virtual machine, or during autolog processing.

The CP accounting function uses general-purpose register 1 for input to entry point HCPACUON. Register 1 contains the address of a parameter list, called AONPARM, that is described in [Figure 12 on page 167](#). The DSECT for this parameter list must be preserved as shown.

```

AONPARM  DSECT ,
****    AONPARM -
*
*
*      +-----+
*      |          AONBUFF          |
*      +-----+
*      8
*
****    AONPARM -
*****
*
* Parameters for IBM 3277 Operator Identification Card
* Reader Feature.
*
*****
AONBUFF DC    F'0'          Contains one of the following:
*
*                               - Address of a buffer containing
*                               up to 130 bytes of data that
*                               has been read from the IBM 3277
*                               Operator Identification Card
*                               Reader Feature. The data is
*                               user-defined for terminal
*                               operator identification.
*
*                               - 0, if the reader feature was
*                               not available
*
*                               - 4, if an unsuccessful attempt
*                               to read this reader feature
*                               was made.
*
AONSIZE  EQU    (*-AONPARM+7)/8      AONPARM size in doublewords

```

Figure 12. AONPARM Parameter List for Entry Point HCPACUON

Upon return to the CP accounting function from entry point HCPACUON, general-purpose register 15 must contain one of the following return codes:

Value	Meaning
X'00000000'	Normal processing continues.
X'00000004'	The terminal operator should be forced off the system. However, if the terminal operator was automatically logged on during initialization, you cannot force the operator off the system.

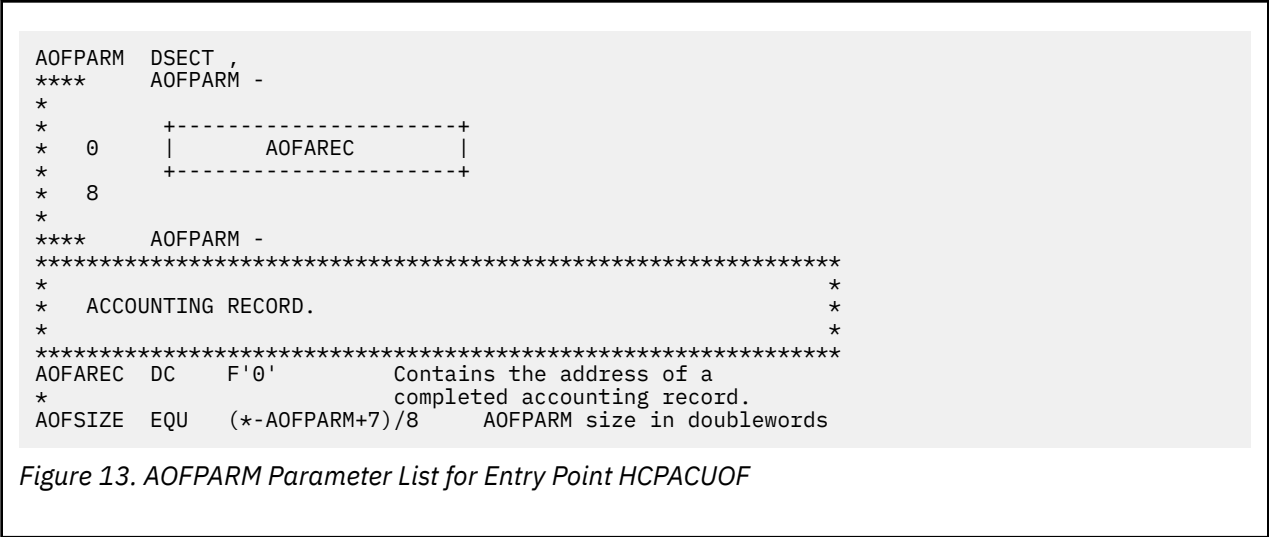
Entry Point HCPACUOF

HCPACUOF supports the addition of installation-supplied code for examining and modifying accounting records. HCPACUOF is called when any of the following occurs:

- A virtual machine operator or system operator enters a LOGOFF or FORCE command
- The system operator enters an ACNT or SET ACCOUNT command
- CP starts the checkpointing function
- A virtual machine issues DIAGNOSE code X'4C' to create a user-initiated accounting record.

For the first three conditions listed above, CP generates accounting record type 1, 2, 3, B, or C. For the last condition, CP generates type C0 accounting records. HCPACUOF is not called during the generation of other types of accounting records.

The CP accounting function uses general-purpose register 1 for input to entry point HCPACUOF. Register 1 contains the address of a parameter list, called AOFPARM, that is described in [Figure 13 on page 168](#). The DSECT for this parameter list must be preserved as shown.



Upon return to the CP accounting function from entry point HCPACUOF, general-purpose register 15 must contain following return code:

Value	Meaning
X'00000000'	Normal processing continues

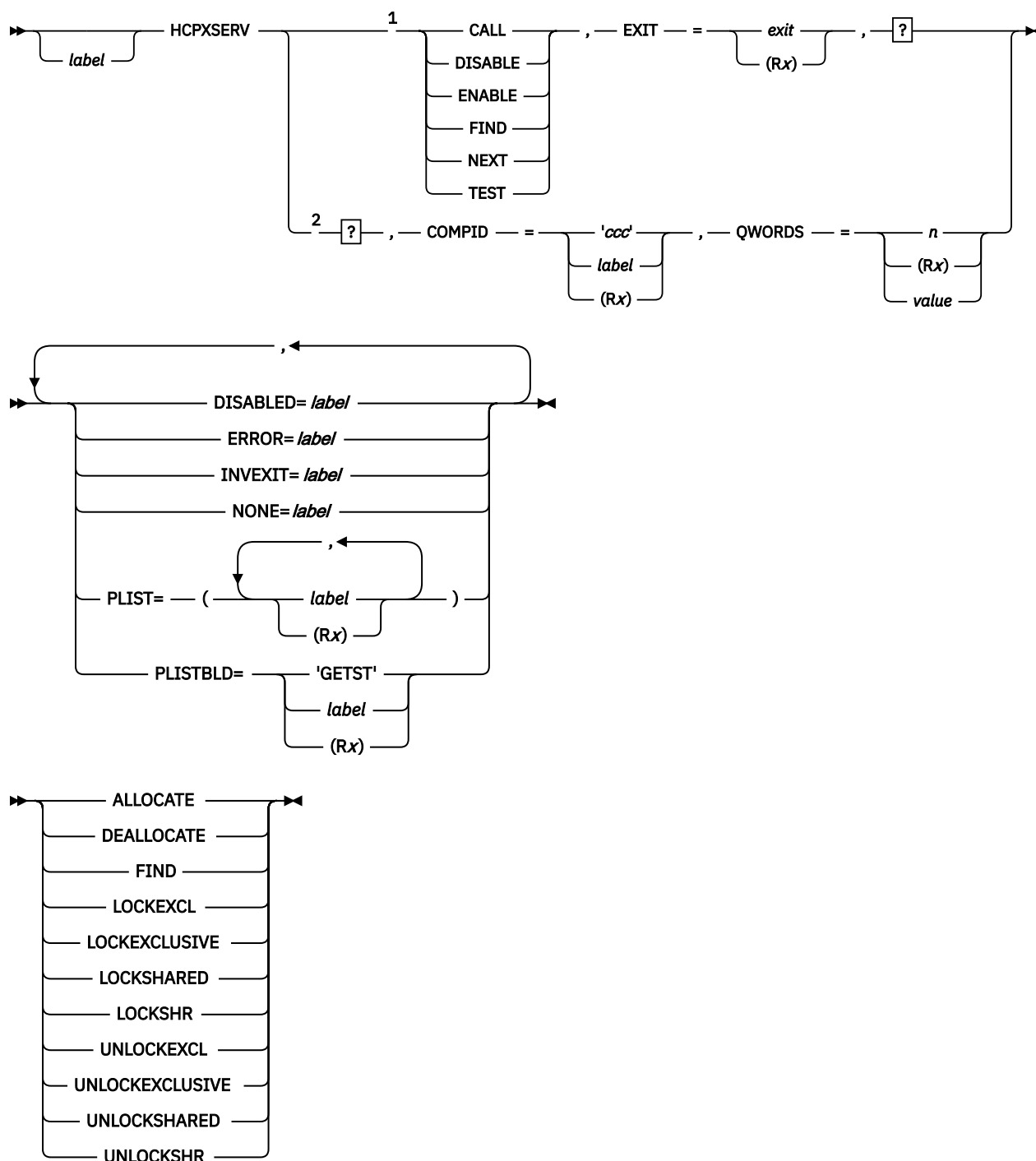
Appendix F. CP Exit Macros

This section describes the CP exit macros that are used with CP exit points. The CP exit macros are:

Macro	Function	Page
HCPXSERV	performs any of the following tasks: <ul style="list-style-type: none"> • Locate exit control blocks • Enable or disable or test exits • Call exit routines • Locate component ID blocks (CMPBKs) • Allocate or deallocate component ID blocks (CMPBKs) • Control access to component ID blocks (CMPBKs). 	“HCPXSERV: CP Exit Services Director” on page 170
MDLATENT	defines the attributes of CP modules and their entry points. The HCPCALL macro uses this information when generating a CP load list.	“MDLATENT: MDLAT Entry Definition” on page 180
MDLATHDR	marks the beginning of a CP module attribute list. This macro initializes variables that will be used and updated by subsequent MDLATENT macro invocations.	“MDLATHDR: MDLAT Header” on page 186
MDLATTLR	marks the end of a CP module attribute list. This macro modifies macro variables to indicate completion of a CP module attribute list.	“MDLATTLR: MDLAT Trailer” on page 187

For information about how to read syntax diagrams, see [“Syntax, Message, and Response Conventions” on page xiii](#).

HCPXSERV: CP Exit Services Director



Notes:

¹ For a list of legal and illegal combinations of EXIT actions and parameters, see [Table 7 on page 174](#).

² For a list of legal and illegal combinations of COMPID actions and parameters, see [Table 7 on page 174](#).

Purpose

Use HCPXSERV to perform any of the following tasks:

- Locate CP exit control blocks
- Enable or disable or test CP exit points
- Call CP exit routines
- Locate component ID blocks (CMPBKs)
- Allocate or deallocate component ID blocks (CMPBKs)
- Control access to component ID blocks (CMPBKs).

Operands

label

is an optional assembler label.

CALL

tells HCPXSERV to generate instructions (a macro expansion) which call the CP exit routines associated with the specified CP exit number.

DISABLE

tells HCPXSERV to generate instructions (a macro expansion) which call the CP exit routine that marks CP exit points as "disabled".

ENABLE

tells HCPXSERV to generate instructions (a macro expansion) which call the CP exit routine that marks CP exit points as "enabled".

FIND

tells HCPXSERV to generate instructions (a macro expansion) which call the CP exit routine that locates the CP exit control block (XITBK) for the specified CP exit number.

NEXT

tells HCPXSERV to generate instructions (a macro expansion) which call the CP exit routine that locates the CP exit control block (XITBK) for the next CP exit number. That is, the CP exit routine looks for the first XITBK with a CP exit number greater than the specified CP exit number. (Remember: a CP exit number only has an XITBK if it is defined and has one or more entry points associated with it.)

TEST

tells HCPXSERV to generate instructions (a macro expansion) which test whether the specified CP exit number is marked as "enabled".

EXIT=*exit*

tells HCPXSERV the CP exit number for which you are generating instructions (a macro expansion). The variable *exit* must be a hexadecimal number in the range X'0000' through X'FFFF'.

EXIT=(*Rx*)

tells HCPXSERV to look in the specified register to find the CP exit number for which you are generating instructions (a macro expansion). The variable *x* must be a decimal number in the range 0 through 15.

DISABLED=*label*

tells HCPXSERV to branch to the specified label if the CP exit point is marked as "disabled".

ERROR=*label*

tells HCPXSERV to branch to the specified label if an error condition is detected (based on the value that is returned in R15).

INVEXIT=*label*

tells HCPXSERV to branch to the specified label if the CP exit number is not valid. That is, the CP exit number is not a 4-digit hexadecimal number in the range X'0000' through X'FFFF'.

NONE=*label*

tells HCPXSERV to branch to the specified label if the CP exit number has no CP exit control block (XITBK). That is, having no XITBK implies that no one used the ASSOCIATE EXIT command or configuration file statement to associate one or more entry points with this CP exit number.

PLIST=(label)

PLIST=(label, label, ...)

PLIST=((Rx),...)

lists labels of data items or registers. Labels and registers may be included in the same parameter list. CP places the addresses of labels and the addresses contained in registers in the parameter list (PLIST) into the parameter list build area (PLISTBLD).

Using the PLIST operand causes the HCPXSERV macro expansion to turn on the high order bit of the last address in the parameter list.

We recommend that you use the standard labels for the first three entries of the PLIST. Using the standard entries will assist in the writing of CP exit routines because each CP exit routine can then be written with the standard entries in mind. These standard entries provide working storage for your current CP exit routines, and they provide room for possible additional function in the future.

The first three standard entries are:

'RESERVED'

is the label of an area that provides room for possible additional function in the future.

'USERWORD'

is the label of the User Area, which provides 32 bytes of user words aligned on a doubleword boundary. The first time CP calls the CP exit point, CP initializes these 32 bytes to binary zeros. Whatever any CP exit routine leaves in these 32 bytes are available to the next CP exit routine. After the last CP exit routine returns control to CP, CP releases these 32 bytes.

Your CP exit routine can use the User Area as working storage.

'TRTTABLE'

is the label of the translate-and-test (TRT) table, which provides 256 bytes of user words aligned on a doubleword boundary. The first time CP calls the CP exit point, CP initializes these 256 bytes to binary zeros. Whatever any CP exit routine leaves in these 256 bytes are available to the next CP exit routine. After the last CP exit routine returns control to CP, CP releases these 256 bytes.

Your CP exit routine can use the TRT table as a translate-and-test table, as a translate table, or as any working storage.

PLISTBLD

tells HCPXSERV where to store the addresses of the data items listed on the PLIST operand. There are three ways to indicate the location of the PLIST:

PLISTBLD='GETST'

tells HCPXSERV to:

- Allocate the PLIST build area from free storage,
- Store the addresses of the PLIST data items,
- Call to the associated CP exit routines, and
- Release the storage allocated for the PLIST build area.

This is the most flexible method because HCPXSERV can calculate the required size of the PLIST build area and allocate it dynamically. However, this is not the most efficient method.

PLISTBLD=label

tells HCPXSERV to store the addresses of the PLIST data items in the storage area at the specified label. HCPXSERV assumes that this storage area is large enough to hold all of the PLIST addresses, because HCPXSERV cannot and does not perform any checking.

PLISTBLD=(Rx)

tells HCPXSERV to store the addresses of the PLIST data items in the storage area whose address is in the specified register. HCPXSERV assumes that this storage area is large enough to hold all of the PLIST addresses, because HCPXSERV cannot and does not perform any checking.

ALLOCATE

tells HCPXSERV to generate instructions (a macro expansion) which calls the CP exit routine that allocates or locates the component ID block (CMPBK) for the specified component ID.

DEALLOCATE

tells HCPXSERV to generate instructions (a macro expansion) which calls the CP exit routine that deallocates the component ID block (CMPBK) for the specified component ID.

FIND

tells HCPXSERV to generate instructions (a macro expansion) which calls the CP exit routine that locates the component ID block (CMPBK) for the specified component ID.

LOCKEXCL**LOCKEXCLUSIVE**

tells HCPXSERV to generate instructions (a macro expansion) which calls the CP exit routine that locates the component ID block (CMPBK) for the specified component ID and acquires the CMPBK as "lock exclusive".

LOCKSHARED**LOCKSHR**

tells HCPXSERV to generate instructions (a macro expansion) which calls the CP exit routine that locates the component ID block (CMPBK) for the specified component ID and acquires the CMPBK as "lock shared".

UNLOCKEXCL**UNLOCKEXCLUSIVE**

tells HCPXSERV to generate instructions (a macro expansion) which calls the CP exit routine that relinquishes the component ID block (CMPBK) that was acquired as "lock exclusive" for the specified component ID.

UNLOCKSHARED**UNLOCKSHR**

tells HCPXSERV to generate instructions (a macro expansion) which calls the CP exit routine that relinquishes the component ID block (CMPBK) that was acquired as "lock shared" for the specified component ID.

COMPID

tells HCPXSERV the component ID for which you are generating instructions (a macro expansion). The component ID is a 3-character string. Because you will probably use the same component ID for the HCPCMPID macro or for the alternate MDLAT macros, we recommend that you use a component ID composed of alphabetic characters (A through Z) and numeric characters (0 through 9), starting with an alphabetic character.

Specify the component ID in any of the following forms:

COMPID='ccc'

defines the 3-character component ID to HCPXSERV.

COMPID=label

tells HCPXSERV to look for the 3-character component ID in the storage location specified by *label*.

COMPID=(Rx)

tells HCPXSERV to look for the 3-character component ID in the storage location pointed to by the address in the specified register.

QWORDS

tells HCPXSERV the number of additional quadwords (units of 16 bytes) to use when allocating a new component ID block (CMPBK).

Specify the additional quadwords in any of the following forms:

QWORDS=n

defines the number of additional quadwords that HCPXSERV should use when allocating the new CMPBK. The variable *n* is a decimal number between 0 and 240. If you omit *n* or specify *n* as 0 (zero), HCPXSERV allocates a CMPBK of the standard size.

QWORDS=(Rx)

tells HCPXSERV to look in the specified register for the number of additional quadwords to use when allocating the new CMPBK.

QWORDS=value

defines the number of additional quadwords in the form of an absolute value. For example:

```
..., QWORDS=LAB2-LAB1
```

Examples**Usage notes**

1. The following table shows which combinations of EXIT and COMPID actions and parameters are valid:

Table 7. Legal and Illegal HCPXSERV Action/Parameter Combinations									
EXIT Action	Parameter								
	EXIT	DISABLED	ERROR	INVEXIT	NONE	PLIST	PLISTBLD	COMPID	QWORDS
CALL	Req	•	Req	Opt ¹	•	Opt ²	Opt ²	•	•
DISABLE	Req	•	Opt	Opt ¹	Opt	•	•	•	•
ENABLE	Req	•	Opt	Opt ¹	Opt	•	•	•	•
FIND	Req	•	Opt	Opt ¹	Opt	•	•	•	•
NEXT	Req	•	Opt	Opt ¹	Opt	•	•	•	•
TEST	Req	Req	•	Opt ¹	•	•	•	•	•
COMPID Action									
ALLOCATE	•	•	•	•	•	•	•	Req	Opt
DEALLOCATE	•	•	•	•	•	•	•	Req	•
FIND	•	•	•	•	•	•	•	Req	•
LOCK EXCLUSIVE	•	•	•	•	•	•	•	Req	•
LOCK SHARED	•	•	•	•	•	•	•	Req	•
UNLOCK EXCLUSIVE	•	•	•	•	•	•	•	Req	•
UNLOCK SHARED	•	•	•	•	•	•	•	Req	•
Legend: Req Required combination of action and parameter. • Cannot use this combination of action and parameter. Opt Optional combination of action and parameter. Opt¹ If you specify EXIT=(Rx), this is a required combination of action and parameter. If you specify EXIT=exit, this is an optional combination of action and parameter. Opt² Optional combination of action and parameter, except that PLIST requires PLISTBLD. Note: You can specify the FIND action with both EXIT and COMPID, but not with the same combinations of parameters. For this reason, we listed it twice in this table: once under the EXIT actions and once under the COMPID actions.									

2. The HCPXSERV macro will generate the necessary machine instructions to place the data that you specified into the proper registers. For example, if you specified COMPID='ABC', the HCPXSERV macro will expand to cause register 1 to be loaded with the address of the component ID string 'ABC'. In order that you know which registers will be used, so that you do not expect the value in some register to be preserved across the HCPXSERV macro expansion, this table shows you which registers HCPXSERV will use.

<i>Table 8. Register Contents During HCPXSERV Execution</i>					
EXIT Action	Register				
	R0	R1	R2	R14	R15
CALL	CP Exit Number	PLISTBLD Address	Work Area	Linkage Address	Linkage Address
DISABLE	CP Exit Number	—	—	Linkage Address	Linkage Address
ENABLE	CP Exit Number	—	—	Linkage Address	Linkage Address
FIND	CP Exit Number	—	—	Linkage Address	Linkage Address
NEXT	CP Exit Number	—	—	Linkage Address	Linkage Address
TEST	CP Exit Number	—	—	Work Area	Work Area
COMPID Action					
ALLOCATE	—	COMPID Address	QWORDS Value	Linkage Address	Linkage Address
DEALLOCATE	—	COMPID Address	—	Linkage Address	Linkage Address
FIND	—	COMPID Address	—	Linkage Address	Linkage Address
LOCK EXCLUSIVE	—	COMPID Address	—	Linkage Address	Linkage Address
LOCK SHARED	—	COMPID Address	—	Linkage Address	Linkage Address
UNLOCK EXCLUSIVE	—	COMPID Address	—	Linkage Address	Linkage Address
UNLOCK SHARED	—	COMPID Address	—	Linkage Address	Linkage Address

3. Following the execution of the HCPXSERV macro, you will need to examine the return code, which will have been placed into register 15. If the return code indicates that your HCPXSERV request was successful (which will be indicated by a return code value of 0, and sometimes 4), you may find a returned value in certain registers. This table shows possible return codes and possible returned values.

<i>Table 9. Register Contents After HCPXSERV Completion</i>		
EXIT Action	Register	Contents or Meaning
CALL	R15	Contains the return code as set by your CP exit routine.
DISABLE	R15=0	HCPXSERV updated the control blocks.
	R15=4	There is no XITBK for this CP exit number. If it was specified, HCPXSERV transfers control to the label specified on the NONE parameter. If NONE was omitted, HCPXSERV continues execution at the next sequential instruction.
	R15=8	The CP exit number was not valid. HCPXSERV passes control to the label specified on the INVEXIT parameter.
ENABLE	R15=0	HCPXSERV updated the control blocks.
	R15=4	There is no XITBK for this CP exit number. If it was specified, HCPXSERV transfers control to the label specified on the NONE parameter. If NONE was omitted, HCPXSERV continues execution at the next sequential instruction.
	R15=8	The CP exit number was not valid. HCPXSERV passes control to the label specified on the INVEXIT parameter.
	R15=12	The system is in the process of being IPLed and the IPL parameter NOEXITS was specified after the prompt, either by the system operator or by a CP exit routine. CP rejected this attempt to enable a CP exit point because the system has not finished initialization processing. You must wait until after the IPL to enable this CP exit point.
FIND	R15=0	HCPXSERV found the XITBK for the specified CP exit point.
	R1	Contains the address of the newly-found XITBK.
	R15=4	The XITBK for the specified CP exit point does not exist. If you specified the NONE parameter on the HCPXSERV invocation, control passes to the label specified on NONE. If you did not specify NONE, control passes to the next sequential instruction.
	R15=8	the CP exit number was not valid. HCPXSERV passes control to the label specified on the INVEXIT parameter.
NEXT	R15=0	HCPXSERV found the XITBK for the CP exit point after the specified CP exit point.
	R1	Contains the address of the newly-found XITBK.
	R15=4	There is no XITBK after the specified CP exit point. If you specified the NONE parameter on the HCPXSERV invocation, control passes to the label specified on NONE. If you did not specify NONE, control passes to the next sequential instruction.
	R15=8	the CP exit number was not valid. HCPXSERV passes control to the label specified on the INVEXIT parameter.

Table 9. Register Contents After HCPXSERV Completion (continued)

EXIT Action	Register	Contents or Meaning
TEST	—	<p>If the XITBK appears to be defined and enabled, control passes to the next sequential instruction after the HCPXSERV macro.</p> <p>If the XITBK appears to be disabled, control passes to the label specified on the DISABLED parameter.</p> <p>In general, the registers are not changed, with the exception of R0, R14, and R15, whose contents are destroyed after HCPXSERV completes execution.</p>

Table 10. Register Contents After HCPXSERV Completion Continued

COMPID Action	Register	Contents or Meaning
ALLOCATE	R15=0	HCPXSERV successfully allocated the component ID block (CMPBK).
	R1	Contains the address of CMPBK. The CMPBK lock was acquired as exclusive, so the caller is responsible for relinquishing the lock.
	R15=4	The CMPBK could not be allocated as it already exists.
	R1	Contains the address of the existing CMPBK. Because the CMPBK was not allocated with this invocation of HCPXSERV, no lock was acquired.
DEALLOCATE	R15=8	Reserved for IBM use and currently unused.
	R15=12	The QWORDS value in R2 was not valid; no CMPBK was allocated.
	R15=0	HCPXSERV successfully deleted the specified CMPBK.
	R15=4	HCPXSERV could not locate the specified CMPBK for deletion.
FIND	R15=8	Reserved for IBM use and currently unused.
	R15=12	The CMPBK lock was not an exclusive lock.
	R15=0	HCPXSERV successfully found the CMPBK.
	R1	Contains the address of the specified CMPBK.
LOCK EXCLUSIVE	R15=4	HCPXSERV could not find the specified CMPBK.
	R15=8	Reserved for IBM use and currently unused.
	R15=12	The CMPBK lock was destroyed.
	R15=16	There was an unexpected return code from the locking routine (HCPLCKAX).
	R1	Contains the address of the specified CMPBK.

Table 10. Register Contents After HCPXSERV Completion Continued (continued)		
COMPID Action	Register	Contents or Meaning
LOCK SHARED	R15=0 R1	HCPXSERV successfully acquired the shared lock on the specified CMPBK. Contains the address of the specified CMPBK.
	R15=4	HCPXSERV could not find the specified CMPBK; no lock was acquired.
	R15=8	Reserved for IBM use and currently unused.
	R15=12	The CMPBK lock was destroyed.
	R15=16	There was an unexpected return code from the locking routine (HCPLCKAS).
UNLOCK EXCLUSIVE	R15=0	HCPXSERV successfully relinquished the exclusive lock on the specified CMPBK.
	R15=4	HCPXSERV could not find the specified CMPBK; no lock was relinquished.
	R15=8	Reserved for IBM use and currently unused.
	R15=12	The CMPBK lock was destroyed.
	R15=16	There was an unexpected return code from the unlocking routine (HCPLCKRX).
UNLOCK SHARED	R15=0	HCPXSERV successfully relinquished the shared lock on the specified CMPBK.
	R15=4	HCPXSERV could not find the specified CMPBK; no lock was relinquished.
	R15=8	Reserved for IBM use and currently unused.
	R15=12	The CMPBK lock was destroyed.
	R15=16	There was an unexpected return code from the unlocking routine (HCPLCKRS).

Example 1

To locate the CMPBK for your component ID (in this example, "C22") and to acquire shared control over that CMPBK (if it exists), code the following:

```

HCPXSERV  LOCKSHARED,      Locate and lock      X
          COMPID='C22'    ... my Component ID Block

HPCASRC  (GOTIT,          00: All OK              X
          MAKEIT,          04: No such CMPBK       X
          ABEND7,          08: Bad return code?    X
          MAKEIT,          12: CMPBK was destroyed X
          ABEND11),        16: Bad return code?    X
          ELSE=ABEND33     ??: Bad return code?

ABEND7   DS      0H
          HCPABEND 007,HARD

ABEND11  DS      0H
          HCPABEND 011,HARD

ABEND33  DS      0H
          HCPABEND 033,HARD

```

Example 2

To allocate the CMPBK for your component ID ("TAS"), acquire exclusive control over that CMPBK (if it does not yet exist), and define an additional 32 bytes (2 quadwords) of CMPBK space, code the following:

```

HCPXSERV  ALLOCATE,      Allocate and lock      X
           COMPID='TAS', ... my Component ID Block X
           QWORDS=2
HPCPCASRC (GOTIT,        00: All OK              X
           BEENDONE,      04: Already exists      X
           ABEND1,         08: Bad R1 value?      X
           ABEND2),        12: Bad R2 value?      X
           ELSE=ABEND33    ??: Bad return code?
ABEND1    DS      0H
           HCPABEND 001,HARD
ABEND2    DS      0H
           HCPABEND 002,HARD
ABEND33   DS      0H
           HCPABEND 033,HARD

```

Example 3

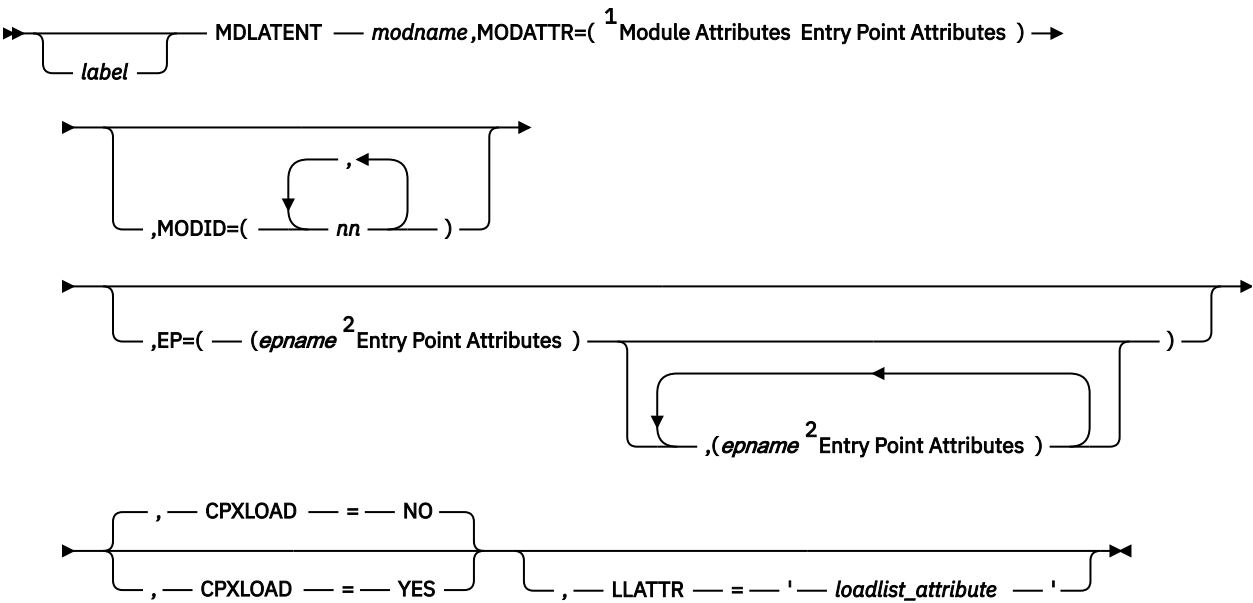
To call the CP exit routines for your CP exit number (in this example, "FEED"), have HCPXSERV allocate and release the parameter list (PLIST) build area, and pass the standard PLIST entries and a control block named C22BK, code the following:

```

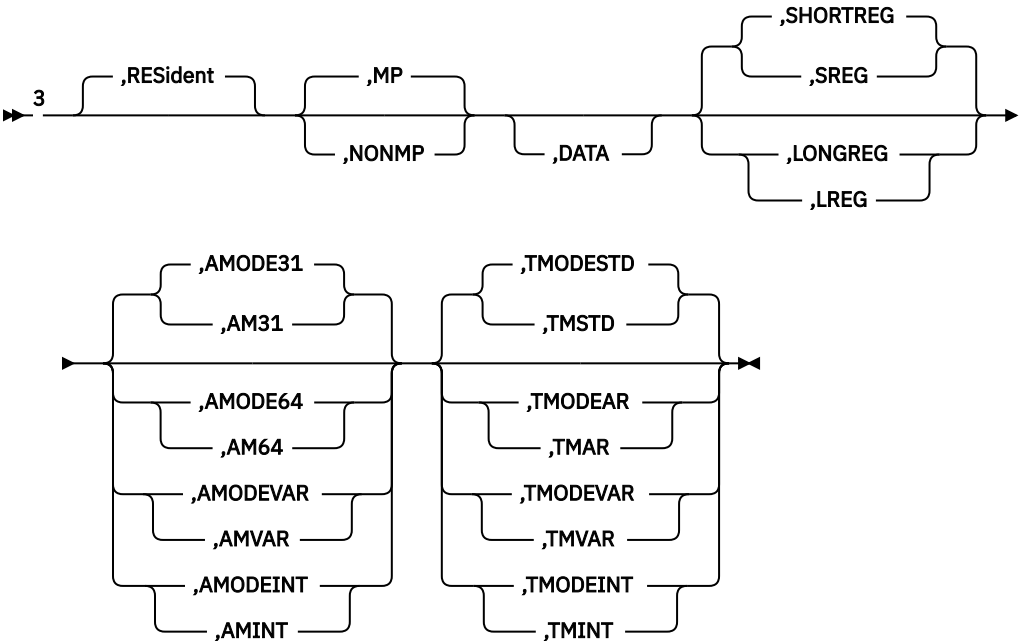
HCPXSERV  CALL,          Call my exit FEED      X
           EXIT=FEED,      X
           PLIST=('RESERVED', X
           'USERWORD',     X
           'TRTTABLE',     X
           C22BK),         X
           PLISTBLD='GETST', X
           ERROR=WHAT_HAPPEN
:
WHAT_HAPPEN DS  0H

```

MDLATENT: MDLAT Entry Definition



Module Attributes

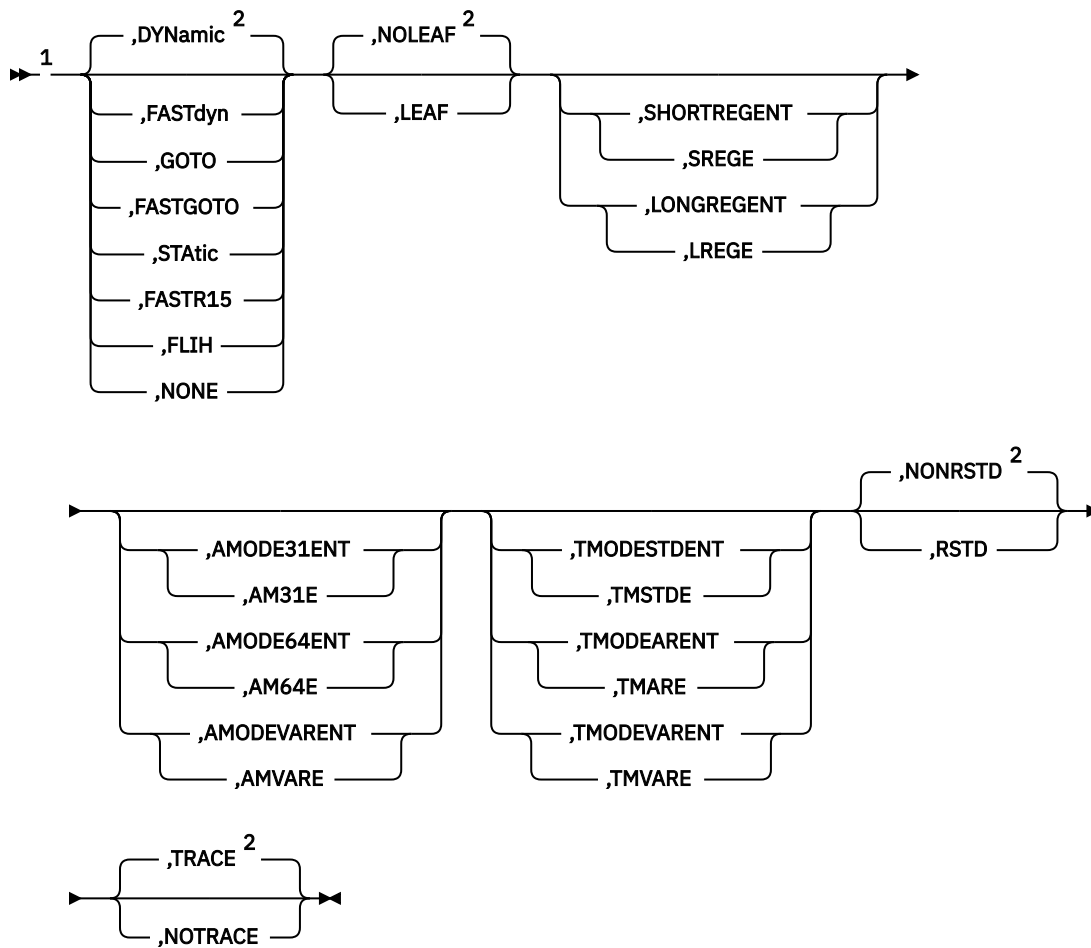


Notes:

- ¹ At least one module attribute or entry point attribute must be specified. The separator (,) must be omitted before the first attribute in the string.
- ² At least one entry point attribute must be specified.
- ³ Module attributes may be specified in any order.



Entry Point Attributes



Notes:

- ¹ Entry point attributes may be specified in any order.
² Default only on MODATTR, not on EP.

Purpose

Use MDLATENT to define the attributes of CP modules and their entry points. The HCPCALL macro uses this information when generating a CP load list.

Operands

label

is an optional assembler label.

modname

is the 6-character name of the module that CP should use when generating the load list.

MODATTR=

defines the attributes for the CP module specified by *modname*. At least one module attribute or entry point attribute must be specified on this parameter. You can specify the attributes in any order.

Most module attributes imply corresponding entry point attributes, as indicated in the following table. Other entry point attributes also have defaults on MODATTR. Entry point attributes implied or specified on MODATTR are the default attributes for all entry points in the module unless overridden. You can override an implied default entry point attribute by specifying a different default entry point attribute on MODATTR or by specifying attributes for specific entry points on the EP parameter. If a module attribute does not imply a corresponding default entry point attribute, you must specify the

appropriate default entry point attribute on MODATTR. You can then override that default for specific entry points on the EP parameter. Entry point attributes are described under the EP parameter.

Module Attribute	Implied Default Entry Point Attribute
AMODE31	AMODE31ENT
AMODE64	AMODE64ENT
AMODEVAR	(none)
AMODEINT	AMODE31ENT
TMODESTD	TMODESTDENT
TMODEAR	TMODEARENT
TMODEVAR	(none)
TMODEINT	TMODESTDENT
SHORTREG	SHORTREAGENT
LONGREG	LONGREAGENT

The valid module attributes are:

RESident

tells CP that the specified module is resident. Because all CP modules are resident, this attribute is supported as a default for compatibility; it should not be specified.

MP

tells CP that the specified module can run on any processor. This is the default.

NONMP

tells CP that the specified module must run on the master processor.

DATA

tells CP that the specified module is a data or work area module.

SHORTREG

SREG

tells CP that this module uses 32-bit registers. This is the default.

The SHORTREG module attribute implies a default entry point attribute of SHORTREAGENT.

LONGREG

LREG

tells CP that this module uses 64-bit registers.

The LONGREG module attribute implies a default entry point attribute of LONGREAGENT.

AMODE31

AM31

tells CP that this module will run in 31-bit addressing mode. This is the default.

The AMODE31 module attribute implies a default entry point attribute of AMODE31ENT.

AMODE64

AM64

tells CP that this module will run in 64-bit addressing mode.

The AMODE64 module attribute implies a default entry point attribute of AMODE64ENT.

AMODEVAR

AMVAR

tells CP that this module will run in 31-bit addressing mode or 64-bit addressing mode, unpredictably.

The AMODEVAR module attribute does not imply a default entry point attribute. Therefore you must specify the default addressing mode entry point attribute on MODATTR.

AMODEINT

AMINT

tells CP that this module will be in 31-bit addressing mode at every macro.

The AMODEINT module attribute implies a default entry point attribute of AMODE31ENT.

TMODESTD

TMSTD

tells CP that this module will run in Primary Space mode. This is the default.

The TMODESTD module attribute implies a default entry point attribute of TMODESTDENT.

TMODEAR

TMAR

tells CP that this module will run in Access Register mode.

The TMODEAR module attribute implies a default entry point attribute of TMODEARENT.

TMODEVAR

TMVAR

tells CP that this module will run in Primary Space mode or Access Register mode, unpredictably.

The TMODEVAR module attribute does not imply a default entry point attribute. Therefore you must specify the default translation mode entry point attribute on MODATTR.

TMODEINT

TMINT

tells CP that this module will be in Primary Space mode at every macro.

The TMODEINT module attribute implies a default entry point attribute of TMODESTDENT.

MODID=(nn)

MODID=(nn,nn)

MODID=(nn,nn,nn)

are the persistent numeric identifiers that can be used as nicknames to represent this module in control blocks and trace entries. The variable *nn* is the value from &HCPNXTID. When creating a new MODID, use the value currently specified for the &HCPNXTID variable in the SETNXTID macro. The SETNXTID macro must also be updated so that the value assigned to &HCPNXTID is one more than the value used for highest MODID that is being used by all modules in the system. One module can have a maximum of three module IDs (*nn*).

EP=

defines the attributes for specific entry points in this module. For each entry point, you can specify these attributes in any order.

Attributes specified on the EP parameter override entry point attributes specified or implied on the MODATTR parameter. However, only the attributes specified on EP override MODATTR attributes. Unless overridden by an attribute specified here, the MODATTR entry point attribute (explicit or defaulted) remains in effect.

The valid entry point attributes are:

epname

is the name of an entry point in this CP module. Each *epname* must be a string of 1 to 8 characters. The first character must be alphabetic or one of the following special characters: dollar sign (\$), number sign (#), underscore (_), or at sign (@). The rest of the string can be alphanumeric characters, the four special characters (\$, #, _, and @), or any combination thereof.

DYNamic

tells CP that this entry point is called with a dynamic savearea. This is the default on MODATTR.

FASTdyn

tells CP that this entry point will be entered using a faster HCPCALL that provides fewer services, such as no processor switching and fewer validity checks. SAVEWRK and SVGWRK are not cleared.

GOTO

tells CP that this entry point will be entered using an HCPGOTO statement that establishes the module's base register.

FASTGOTO

tells CP that this entry point will be entered using a faster HCPGOTO that provides fewer services, such as no processor switching.

STATIC

tells CP that this entry point is called with a static savearea.

FASTR15

is a special version of the FASTGOTO attribute. It is intended for IBM use only.

FLIH

identifies the entry point as a First Level Interrupt Handler. When a hardware interrupt event occurs and a PSW swap takes place, an FLIH entry point is where execution resumes. This attribute is not intended for general CP exit linkage.

NONE

tells CP that this entry point does not have a savearea to use.

NOLEAF

tells CP that instruction execution reduction is not in effect. This is the default on MODATTR.

LEAF

can be used with STATIC entry points to reduce instruction execution. STATIC entry points may save Access Registers or may skip saving ARs. There are extra instructions executed in calling a subroutine, and returning from a subroutine, that make the right runtime decision. If the programmer knows that saving ARs is not necessary in the subroutine, and that the subroutine never calls anything else where AR-saving is needed, the programmer can assign the LEAF attribute to the entry point to avoid the execution of the extra instructions. This situation is typically used only in a high performance execution path, where the programmer wants to reduce instruction execution as much as possible.

SHORTREGENT

SREGE

tells CP that this entry point uses 32-bit registers.

LONGREGENT

LREGE

tells CP that this entry point uses 64-bit registers.

AMODE31ENT

AM31E

tells CP that this entry point will be entered in 31-bit addressing mode.

AMODE64ENT

AM64E

tells CP that this entry point will be entered in 64-bit addressing mode.

AMODEVARENT

AMVARE

tells CP that this entry point will be entered in whatever addressing mode is running.

TMODESTDENT

TMSTDE

tells CP that this entry point will be entered in Primary Space mode or left in Home Space mode.

TMODEARENT

TMARE

tells CP that this entry point will be entered in Access Register mode.

TMODEVARENT

TMVARE

tells CP that this entry point will be entered in whatever translation mode is running.

NONRSTD

tells CP that this is a nonrestricted entry point. That is, this entry point can be called directly using HCPCALL. This is the default on MODATTR.

RSTD

tells CP that this is a restricted entry point. That is, this entry point can be called only by using a special-purpose cover macro.

TRACE

tells CP to trace the call and return for dynamic linkage, unless overridden by specifying TRACE=NO on the HCPCALL macro call. This is the default on MODATTR.

NOTRACE

tells CP not to trace the call and return for dynamic linkage, unless overridden by specifying TRACE=YES on the HCPCALL macro call.

CPXLOAD=NO

tells CP that this module will not be loaded using a CPXLOAD command or configuration file statement. This means that the module will be included in the CP load list. This is the default.

CPXLOAD=YES

tells CP that this module will loaded using a CPXLOAD command or configuration file statement. This means that the module will not be included in the CP load list.

LLATTR='loadlist_attribute'

provides CP with any additional information to be added to the PUNCH statement of the CP load list. The variable *loadlist_attribute* must be enclosed in single quotation marks. Because the information is enclosed in single quotes, you can specify any combination of alphabetic, numeric, and special characters (blanks, parentheses, and so forth). For example, you would specify LLATTR=' (LANG ' if this module was a message repository. Normal assembler rules apply to the specification of single quotes as data for this operand. For example, if you specify two single quotes immediately adjacent to each other, the first single quote is not treated as the end of string delimiter. Instead, a single quote is placed in the string at that point.

Note: The PUNCH statement cannot exceed 80 characters, including this additional information. If the PUNCH statement does exceed 80 characters, you will receive an assembler error and the statement will be ignored.

Usage notes

1. This macro defines attributes of modules and entry points within a module attribute list. It must follow a MDLATHDR invocation and precede a MDLATTLR invocation for the list.
2. Entry points called indirectly should be defined with attributes DYN, AM31E, SREGE, and TMSTDE. Entry points called directly may be defined with attributes AM64E, LREGE, and so on.

Entry points called indirectly would include:

- Dynamically defined CP commands
- Dynamically defined CP Diagnose code routines
- Dynamically defined CP exit routines

The reason for the attribute restriction on indirect calls is that indirect calls are handled by service routines that are themselves AM31E, SREGE, and TMSTDE, and will not support the broader attributes. However, if the modules loaded by CPXLOAD can be called directly (for example, if they are all part of the same CPXLOAD operation or were loaded by CPXLOAD PERMANENT), then they may use direct calls among themselves and may be defined with the broader attributes like AM64, LREG, TMAR, and so on.

3. For more information and examples of how to use the MDLATENT macro, see [Appendix I, “Updating the CP Load List,”](#) on page 213.

MDLATHDR: MDLAT Header



Purpose

Use MDLATHDR to mark the beginning of a CP module attribute list. This macro initializes variables that will be used and updated by subsequent MDLATENT macro invocations.

Operands

label

is an optional assembler label.

epname

is the symbolic name of the first positional parameter on the alternate MDLAT macro (xxxMDLAT MACRO, where xxx is the 3-character component ID) in which this MDLATHDR macro is used. This name should appear exactly as coded on the alternate MDLAT macro.

For example, if you code a macro variable of "&EPNAME" on the on the alternate MDLAT macro definition line, then you must specify "&EPNAME" for this operand, like this:

```
&LABEL      MACRO
              xxxMDLAT &EPNAME
              MDLATHDR &EPNAME
```

Usage notes

- 1. This macro marks the beginning of a CP module attribute list. It must precede any MDLATENT macro invocations for the list. The MDLATTLR macro marks the end of the CP module attribute list.
- 2. For more information and examples of the use of this macro, see [Appendix I, “Updating the CP Load List,”](#) on page 213.

MDLATTLR: MDLAT Trailer



Purpose

Use MDLATTLR to mark the end of a CP module attribute list. This macro modifies macro variables to indicate completion of a CP module attribute list.

Operands

label

specifies an optional assembler label.

Usage notes

1. This macro marks the end of a CP module attribute list. It must follow any MDLATENT macro invocations for the list. The MDLATHDR macro marks the beginning of the CP module attribute list.
2. For more information and examples of the use of this macro, see [Appendix I, “Updating the CP Load List,”](#) on page 213.

Appendix G. CP Parser Macros and Routines

This section describes the CP Parser related macros, and the calling conventions and parameter lists for CP Parser related routines.

The Parser related macros are:

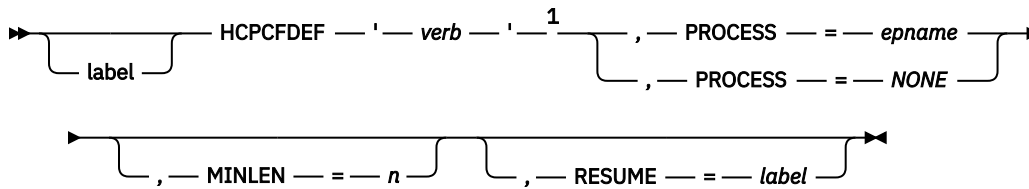
Macro	Function	Page
HCPCFDEF	denotes the initial state for a given command or configuration file statement.	“HCPCFDEF: Command/Config File Statement Definition Macro” on page 190
HCPDOSYN	generates the necessary data structures to describe the syntax.	“HCPDOSYN: Parser Syntax Table Generator Macro” on page 191
HCPSTDEF	denotes additional states that are possible.	“HCPSTDEF: Parser State Definition Macro” on page 193
HCPTKDEF	defines the transition rules from one state to another.	“HCPTKDEF: Parser Token Definition Macro” on page 196

The Parser related routines are:

Routine	Function	Page
HCPZPRPC	parses the remainder of the command line.	“HCPZPRPC – Parse Remainder of Command Line” on page 205
HCPZPRPG	parses an arbitrary General System Data Block (GSDBK).	“HCPZPRPG – Parse Any GSDBK” on page 206
Pre-Processor	handles preprocessing of a command or statement prior to being processed by the parser. This routine is invoked as the result of being specified as part of the PREPROC parameter of the HCPDOSYN macro.	“Pre-Processing Routines” on page 206
Post-Processor	handles processing after all processing has been successfully completed by the parser for a command or statement. This routine is invoked as the result of being specified as part of the PROCESS parameter of the HCPCFDEF macro.	“Post-Processing Routines” on page 207

For information about how to read syntax diagrams, see [“Syntax, Message, and Response Conventions” on page xiii](#).

HCPCFDEF: Command/Config File Statement Definition Macro



Notes:

¹ You can specify the following operands in any order.

Purpose

Use the HCPCFDEF macro to define the initial state for a given CP command or configuration file statement.

Operands

label

is an optional assembler label.

verb

is the statement or command verb. The transition to lower case defines the minimum abbreviation. The statement or command verb must be enclosed in single quotes.

PROCESS=epname

is the label of an entry point used to process the result of an input string scan. For coding details of the entry point, such as necessary register entry and exit conditions, refer to [“Post-Processing Routines”](#) on page 207.

PROCESS=NONE

indicates that the results of parsing should not be processed further.

MINLEN=n

is the minimum abbreviation that can be used to specify the verb. If MINLEN is not specified, the minimum abbreviation length is determined by the transition from upper to lower case in the verb.

RESUME=label

is the label of an HCPSTDEF entry where parsing is to resume. If RESUME is not specified, the parser continues processing with the first HCPSTDEF following the HCPCFDEF.

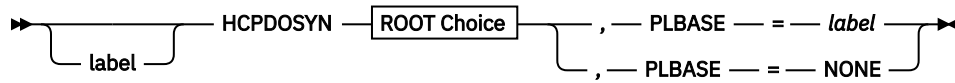
Examples

There are times when you may not wish to begin parsing the command from the beginning of the command GSDBK. In this example, "QUERY" and "CPCMDs" would have been processed by the command router in order to determine which syntax table to use in processing the command. Therefore, we want to resume parser processing from the point where the command router left off.

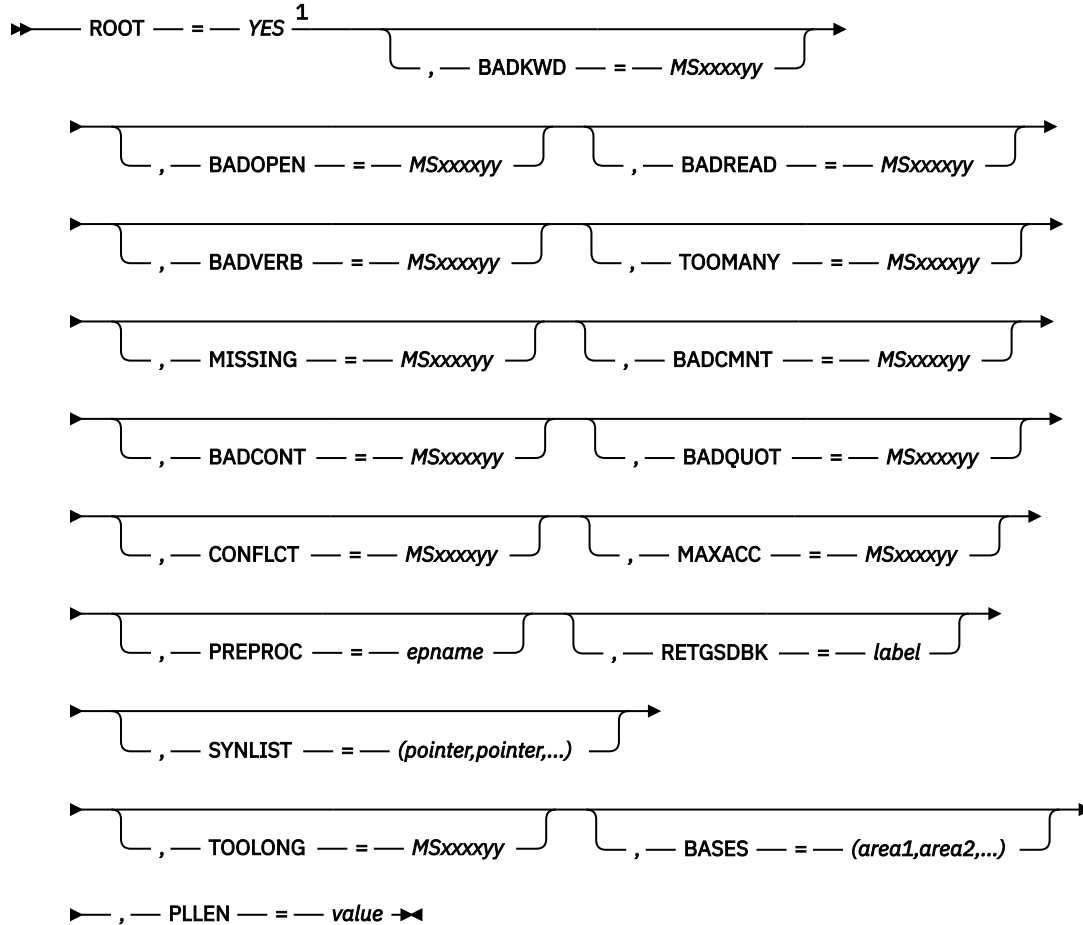
```
HCPLCSQC HCPCFDEF 'Query',RESUME=QRESUME,PROCESS=NONE
*
    HCPSTDEF REQMATCH=YES
    HCPTKDEF 'CPCMDs'

*
QRESUME HCPSTDEF REQMATCH=YES
.
```

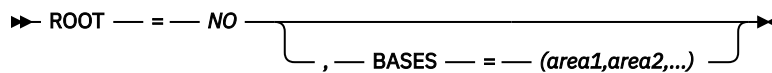
HCPDOSYN: Parser Syntax Table Generator Macro



ROOT Choice 1



ROOT Choice 2



Notes:

¹ You can specify the following operands in any order.

Purpose

Generates syntax table defined via previous invocations of HCPCFDEF, HCPSTDEF and HCPTKDEF macros.

Operands

label

is an optional assembler label.

ROOT=YES

indicates that table being defined is the root of a syntax table. This causes additional control structure to be set up so that the information can be used as a syntax table.

ROOT=NO

indicates that table being defined is not the root of a syntax table.

PLBASE=label

is the label of the base output plist.

PLBASE=NONE

indicates that a base output plist is not present.

PLLEN=n

is the length of the base output plist in bytes. This operand is required if PLBASE=NONE is not specified.

SYNLIST=(pointer,pointer,...)

is a list of address of other branches of the syntax tree.

BASES=(area1,area2,...,arean)

is a list of names of data areas whose addresses will be passed to the parser on invocation.

BADKWD=MSxxxxyy

is a the equate of the message to be issued for a bad keyword error.

BADOPEN=MSxxxxyy

is a the equate of the message to be issued when an error is encountered opening a configuration file.

BADREAD=MSxxxxyy

is a the equate of the message to be issued when an error is encountered reading a configuration file.

BADVERB=MSxxxxyy

is a the equate of the message to be issued for a bad statement or command error.

TOOMANY=MSxxxxyy

is a the equate of the message to be issued for an extra operands on the end of the line error.

MISSING=MSxxxxyy

is a the equate of the message to be issued for a missing keyword error.

BADCMNT=MSxxxxyy

is a the equate of the message to be issued for a bad REXX comment error.

BADCONT=MSxxxxyy

is a the equate of the message to be issued for a bad continuation error.

TOOLONG=MSxxxxyy

is a the equate of the message to be issued for a records > 4000 error.

CONFLCT=MSxxxxyy

is a the equate of the message to be issued for a conflicting options error.

MAXACC=MSxxxxyy

is a the equate of the message to be issued for a reached maximum accumulations error.

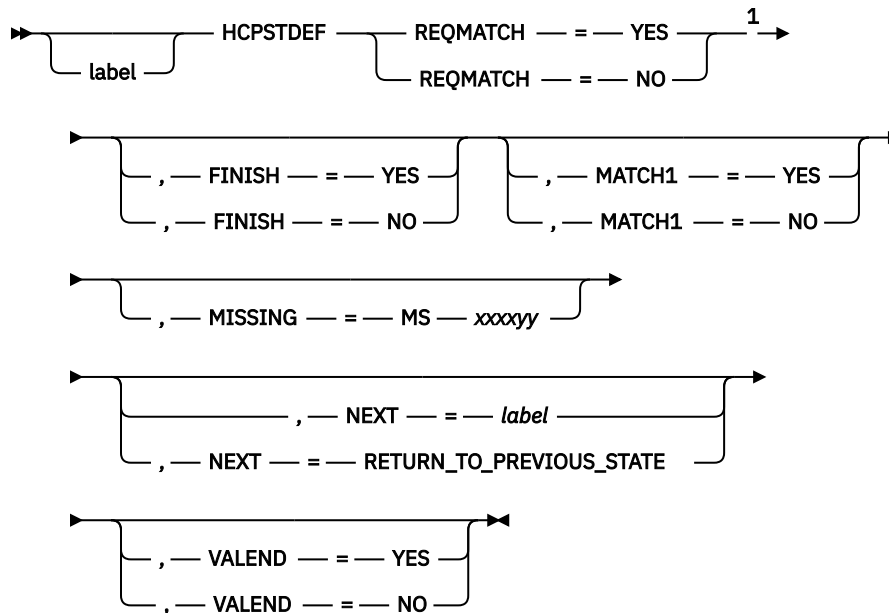
PREPROC=epname

indicates a routine to be called by the parser prior to processing the input.

RETGSDBK=label

label of the fullword field which should point to the queue of error message GSDBKs.

HCPSTDEF: Parser State Definition Macro



Notes:

¹ You can specify the following operands in any order.

Purpose

Use the HCPSTDEF macro to define a state in the finite state machine that expresses a command or statement syntax.

Operands

label

is an optional assembler label.

REQMATCH=YES

REQMATCH=NO

tells the parser whether a token match is required. If CP does not find a match among the HCPTKDEFs after the HCPSTDEF, REQMATCH=NO allows a transition to the next state. This operand is required and has no default.

FINISH=YES

FINISH=NO

tells the parser whether transition to the next state is allowed once this state has been processed. FINISH=YES means no additional state transitions are allowed. FINISH=NO means additional state transitions are allowed. If FINISH is not specified and the HCPSTDEF macro is the final HCPSTDEF macro before an HCPCFDEF or an HCPDOSYN macro, the default is FINISH=YES. If FINISH is not specified and the HCPSTDEF macro is not the final HCPSTDEF macro before an HCPCFDEF or an HCPDOSYN macro, the default is FINISH=NO.

MATCH1=YES

MATCH1=NO

tells the parser whether there must be at least one match before the REQMATCH=NO and VALEND=YES options are honored. MATCH1=YES forces the choosing of one or more of a list of keywords.

MISSING=MSxxxxyy

tells the parser which error message to issue if a required token is not supplied. If there is no MISSING parameter on the HCPSTDEF macro, then the parser uses the MISSING parameter on the HCPDOSYN macro.

NEXT=label

defines the default transition to the next state. If omitted, the default transition is to the next HCPSTDEF definition (if one follows). Even if specified, you can override the state transition with the NEXT option on a matched HCPTKDEF macro.

NEXT=RETURN_TO_PREVIOUS_STATE

codes a state that acts like a subroutine. In this case, upon leaving the state, the parser returns to the last state processed. If no such state exists — that is, if the state is the first state — no transition takes place.

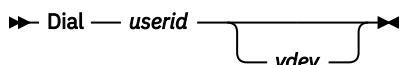
VALEND=YES**VALEND=NO**

tells the parser whether another token is required. VALEND=YES indicates that a state is a valid end state. VALEND=NO indicates that a state is not a valid end state. If VALEND is not specified and the HCPSTDEF macro is the final HCPSTDEF macro before an HCPCFDEF or an HCPDOSYN macro, the default is VALEND=YES. If VALEND is not specified and the HCPSTDEF macro is not the final HCPSTDEF macro before an HCPCFDEF or an HCPDOSYN macro, the default is VALEND=NO.

Examples**Example 1**

This example shows the use of REQMATCH=YES and VALEND=YES.

The syntax of the DIAL command is:



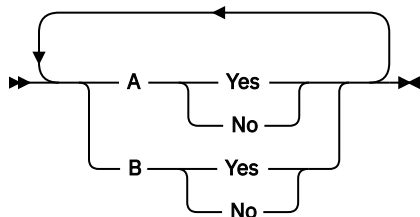
This indicates that "vdev" is optional (in this case, that the command might end) but, if specified, must be a valid hexadecimal virtual device number. The HCPSTDEF and HCPTKDEF macros for this requirement would include these keywords.

```
HCPSTDEF REQMATCH=YES, VALEND=YES
HCPTKDEF TYPE=HEX, RANGE=(0, X'FFFF')
```

Example 2

This example shows the use of NEXT=label.

The parser may need to be directed to continue its work at a HCPSTDEF macro that is not immediately following the one presently being used. For example, this syntax parcel:



cannot be described in a simple "top-down" series of HCPSTDEF and HCPTKDEF macros. Branching to separated macros is necessary. This syntax parcel could be described this way.

```
A_OR_B HCPSTDEF REQMATCH=YES, ...
HCPTKDEF 'A', NEXT=AYN
HCPTKDEF 'B', NEXT=BYN
```

```

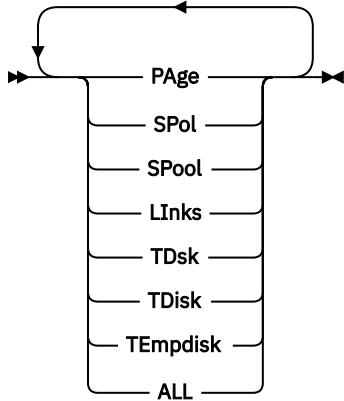
AYN      HCPSTDEF  REQMATCH=YES,NEXT=A_OR_B
          HCPTKDEF  'Yes',ORFLAG=...
          HCPTKDEF  'No',ANDFLAG=...
BYN      HCPSTDEF  REQMATCH=YES,NEXT=A_OR_B
          HCPTKDEF  'Yes',ORFLAG=...
          HCPTKDEF  'No',ANDFLAG=...

```

Example 3

This example shows the use of MATCH1=YES.

The syntax for the DRAIN DASD command ends with a repeated choice of allocation types, at least one of which must be specified.



This syntax parcel will almost work.

```

DRSTOPTS HCPSTDEF  REQMATCH=YES,VALEND=YES,
          NEXT=DRSTOPTS
          HCPTKDEF  'Page',CONFLICT=1,...
          HCPTKDEF  'SPol',CONFLICT=2,...
          HCPTKDEF  'SPool',CONFLICT=2,...
          HCPTKDEF  'LInks',CONFLICT=3,...
          HCPTKDEF  'TDsk',CONFLICT=4,...
          HCPTKDEF  'TDisk',CONFLICT=4,...
          HCPTKDEF  'TEmpdisk',CONFLICT=4,...
          HCPTKDEF  'ALL',CONFLICT=(1,2,3,4),...

```

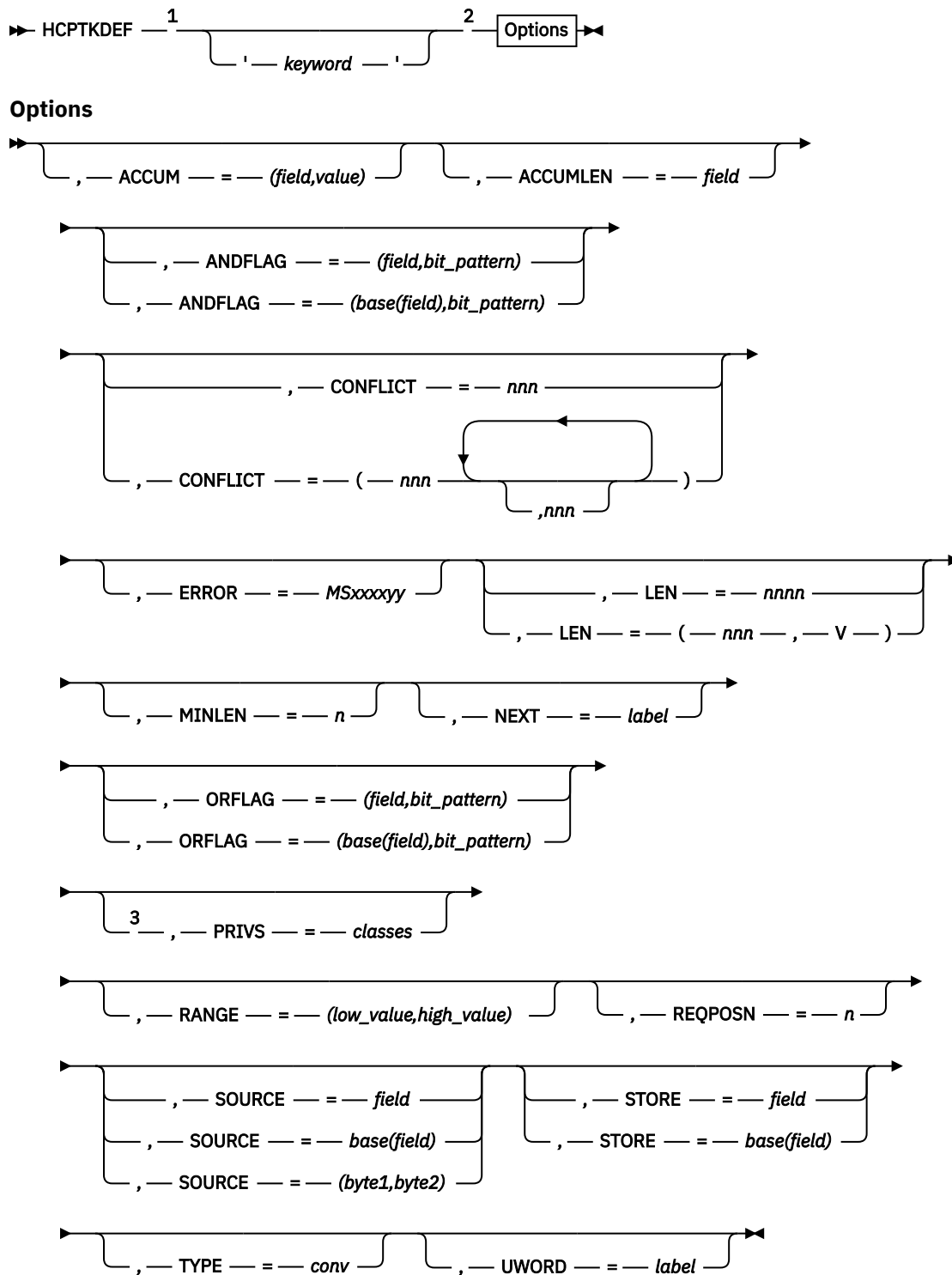
The use of VALEND=YES indicates that the end of input is accepted. Unfortunately, this is also accepted if there was no allocation type specified at all. To require at least 1 to be specified, we need MATCH1=YES.

```

DRSTOPTS HCPSTDEF  REQMATCH=YES,VALEND=YES,MATCH1=YES,
          NEXT=DRSTOPTS
          HCPTKDEF  'Page',CONFLICT=1,...
          HCPTKDEF  'SPol',CONFLICT=2,...
          HCPTKDEF  'SPool',CONFLICT=2,...
          HCPTKDEF  'LInks',CONFLICT=3,...
          HCPTKDEF  'TDsk',CONFLICT=4,...
          HCPTKDEF  'TDisk',CONFLICT=4,...
          HCPTKDEF  'TEmpdisk',CONFLICT=4,...
          HCPTKDEF  'ALL',CONFLICT=(1,2,3,4),...

```

HCPTKDEF: Parser Token Definition Macro



Notes:

- ¹ At a minimum, you must specify either a keyword or the TYPE= option.
- ² You can specify the following options in any order.
- ³ This parameter is only valid for commands, not for configuration file statements.

Purpose

The HCPTKDEF macro contains most of the options that simplify the process of checking syntax and converting data. Each invocation of the macro contains a keyword, a conversion type, or both.

Operands

ACCUM=(*field,value*)

tells the parser that the value stored at the field specified on the STORE= option of the HCPTKDEF macro is an accumulated item. Thus, you may specify more than one such item; if you do, the parser will maintain in the field identified by *field* a halfword count of how many such items it has encountered, and is to consider any attempt to specify more than *value* such items an error. The *field* field must reside in the primary data area used to return information to the calling routine, that is, one identified by the PLBASE keyword on the HCPDOSYN macro. The parser will check that the number of items being accumulated does not exceed the maximum number of items allowed (the value of *value*). The parser will put the converted result of each item being accumulated into the next position in the output field after all previous items that have been accumulated. The parser will not check that there is sufficient storage for all of the items being accumulated.

ACCUMLLEN=*field*

specifies that the accumulated length of varying strings (TYPE=STRING, LEN=(xxx,V)) is to be maintained in the halfword variable identified by *field*.

ANDFLAG=(*field,bit_pattern*)

ANDFLAG=(*base(field),bit_pattern*)

specifies that the flag byte identified by the location (i.e. field or base(field)) is to have a *bit_pattern* ANDed into it.

CONFLICT=*nnn*

CONFLICT=(*nnn[,nnn,]*)

identifies the other operands with which this option conflicts; the variable *nnn* is a decimal value between 1 and 255. List as many numbers as you need on a single CONFLICT keyword.

ERROR=MSxxxxyy

tells the parser which error message to issue if the token is invalid.

LEN=*nnnn*

LEN=(*nnn,V*)

defines the maximum length of a string specification (default is the value of the L' assembler attribute of the STORE= field). LEN can be used to specify that a token of TYPE=STRING is to be stored as a varying character string. If the LEN= value specifies a sublist whose second entry is the character V (for example, LEN=(8,V)) then the token type must be TYPE=STRING. A varying length string is stored with a prefix byte that contains the length of the string value. Varying length strings are not padded to the full length of the string. Normal strings are padded with blanks to the full length of the string.

MINLEN=*n*

tells the parser what minimum abbreviation to accept for a keyword. If omitted, the transition from upper case to mixed case in the specified keyword determines its minimum abbreviation.

NEXT=*label*

tells the parser what state (which HCPSTDEF macro) to enter next. This label overrides any state transition specified on the HCPSTDEF macro or implied by omission of the NEXT keyword on the HCPSTDEF macro.

ORFLAG=(*field,bit_pattern*)

ORFLAG=(*base(field),bit_pattern*)

specifies that the flag byte identified by the location (i.e. field or base(field)) is to have a *bit_pattern* ORed into it.

PRIVS=*classes*

tells the parser that the specified token is restricted to only those users that are authorized for any of the specified IBM class versions of the command. Valid IBM privilege class values are A-G. Specify multiple classes as a single token (for example, AEF). If the user executing the command does not have the required privilege classes for the specified IBM class values, the parser treats the HCPTKDEF

macro as if it did not exist. This operand should only be used for commands and not for configuration file statements.

RANGE=(*low_value*,*high_value*)

specifies the low and high range for a DEC or HEX conversion. Omission of the first number in the ordered pair implies a lower limit of 0, while the omission of the second number implies an upper limit of X'7FFFFFFF'. The default is (0,).

REQPOS=*n*

specifies that the token must be in this position in the input stream. The term "position" in this context is the same as saying that "position *n*" means that we have scanned of *n*-1 blank delimited tokens, so this is the *n*-th blank delimited token. If we are not at this (the value for REQPOS) position in the input stream, then we treat this HCPTKDEF as unmatched.

SOURCE=*field*

SOURCE=*base(field)*

SOURCE=(*byte1*,*byte2*)

tells the parser to move information from the SOURCE location to the location indicated on the STORE parameter. TYPE and SOURCE cannot be specified on the same HCPTKDEF invocation. Both the SOURCE and the STORE location may reside in the primary output data area or any of the data areas specified on the BASES parameter of the HCPDOSYN macro.

Source data may be specified directly in the SOURCE= keyword. Two bytes of data must be specified.

STORE=*field*

STORE=*base(field)*

tells CP where to place the output that results from a conversion or storage reference.

TYPE=*conv*

tells what type of token to expect as the next operand. Additional data verification and processing will be performed on the operand as the result of the type specified. See [Table 11 on page 198](#) for a list of the possible type values.

UWORD=*label*

specifies a value that is to be passed to the post-processing routine.

The valid conversion types are listed in [Table 11 on page 198](#).

Table 11. Conversion types

Conversion Type	Meaning
ACCMODE	accepts a file mode (A-Z) and converts it to a number from 1 to 26. The result is stored in the specified output field.
ADDPRIVS	accepts a single token (with no imbedded blanks) consisting of the plus sign (+) followed by one or more privilege classes (alphanumeric characters in the ranges A through Z and 1 through 6). CP converts the token into a 32-bit bit map where the number 1 marks each privilege class that was specified in the token.
CHAR	a single character conversion of the form X' <i>hh</i> , <i>c</i> , or ' <i>c</i> '. Any character is accepted.

Table 11. Conversion types (continued)

Conversion Type	Meaning
CSECT	<p>accepts and stores a 1- to 8-character token, containing characters only valid in external symbols, in the output field. Valid characters are:</p> <ul style="list-style-type: none"> • Dollar sign (\$), also called currency symbol • Number sign (#), also called pound sign or hash symbol • Commercial "at" sign (@) • Underscore (_) • Alphabetic characters (A through Z) • Numeric characters (0 through 9). <p>These characters are legal in all positions of the token, except for position 1, which does not accept the numeric characters (0 through 9).</p>
DEC	<p>accepts and converts a decimal number. The number must be within the range specified on the RANGE keyword on the HCPTKDEF macro. A range from 1 to 65535 (including both end points) is specified as RANGE=(1,65535). All of the following are also valid ranges: RANGE=(,20), RANGE=(1,), RANGE=(,), RANGE=(X'A',B'1111'), and RANGE=(,EQUATEVAL). The resulting number may be stored in an output field of one, two, three, or four bytes.</p>
DECLIST	<p>accepts a series (or list) of tokens consisting of:</p> <ul style="list-style-type: none"> • Decimal numbers (0 through 9) • Ranges of decimal numbers separated by a dash (–) <p>The number must be within the range specified on the RANGE keyword on the HCPTKDEF macro. A range from 1 to 65535 (including both end points) is specified as RANGE=(1,65535). All of the following are also valid ranges: RANGE=(,20), RANGE=(1,), RANGE=(,), RANGE=(X'A',B'1111'), and RANGE=(,EQUATEVAL).</p>
DECSTOR	<p>accepts a decimal token or a decimal token immediately followed by a non-decimal character. If the token is of the latter form, it is split in two and treated as a decimal number followed by a single character token.</p>
DEVLIST	<p>accepts a series (or list) of tokens consisting of:</p> <ul style="list-style-type: none"> • Hexadecimal numbers (0 through F) • Ranges of hexadecimal numbers separated by a dash (–) <p>The number must be within the range specified on the RANGE keyword on the HCPTKDEF macro. A range from 1 to 65535 (including both end points) is specified as RANGE=(1,65535). All of the following are also valid ranges: RANGE=(,20), RANGE=(1,), RANGE=(,), RANGE=(X'A',B'1111'), and RANGE=(,EQUATEVAL).</p>
DEV RANGE	<p>a single device number or a range of devices addressed in the form xxxx-yyyy, where xxxx and yyyy are both hexadecimal numbers and xxxx is less than yyyy. The converted result is stored as two fullwords, the low device number and the high device numbers. If a single address is specified instead of an address range, the low and high device numbers are identical.</p>

Table 11. Conversion types (continued)

Conversion Type	Meaning
EPNAME	<p>accepts and stores a 1- to 8-character token, containing characters only valid in external symbols, in the output field. Valid characters are:</p> <ul style="list-style-type: none">• Dollar sign (\$), also called currency symbol• Number sign (#), also called pound sign or hash symbol• Commercial "at" sign (@)• Underscore (_)• Alphabetic characters (A through Z)• Numeric characters (0 through 9). <p>These characters are legal in all positions of the token, except for position 1, which does not accept the numeric characters (0 through 9).</p>
EXTRN	<p>accepts and stores a 1- to 8-character token, containing characters only valid in external symbols, in the output field. Valid characters are:</p> <ul style="list-style-type: none">• Dollar sign (\$), also called currency symbol• Number sign (#), also called pound sign or hash symbol• Commercial "at" sign (@)• Underscore (_)• Alphabetic characters (A through Z)• Numeric characters (0 through 9). <p>These characters are legal in all positions of the token, except for position 1, which does not accept the numeric characters (0 through 9).</p>
FILENAME	<p>accepts and stores a 1- to 8-character token, containing characters only valid in CMS file names, in the output field. Valid characters are:</p> <ul style="list-style-type: none">• Dollar sign (\$), also called currency symbol• Number sign (#), also called pound sign or hash symbol• Commercial "at" sign (@)• Underscore (_)• Plus sign (+)• Minus sign or dash (–)• Colon (:)• Alphabetic characters (A through Z)• Numeric characters (0 through 9).

Table 11. Conversion types (continued)

Conversion Type	Meaning
FILETYPE	<p>accepts and stores a 1- to 8-character token, containing characters only valid in CMS file types, in the output field. Valid characters are:</p> <ul style="list-style-type: none"> • Dollar sign (\$), also called currency symbol • Number sign (#), also called pound sign or hash symbol • Commercial "at" sign (@) • Underscore (_) • Plus sign (+) • Minus sign or dash (–) • Colon (:) • Alphabetic characters (A through Z) • Numeric characters (0 through 9).
HEX	accepts and converts a hexadecimal number. As with decimal conversions, the range checks are applied. The resulting number may be stored in an output field of one, two, three, or four bytes.
HEXLIST	<p>accepts a series (or list) of tokens consisting of:</p> <ul style="list-style-type: none"> • Hexadecimal numbers (0 through F) • Ranges of hexadecimal numbers separated by a dash (–) <p>The number must be within the range specified on the RANGE keyword on the HCPTKDEF macro. A range from X'1' to X'FFFF' (including both end points) is specified as RANGE=(1,X'FFFF'). All of the following are also valid ranges: RANGE=(,X'FF'), RANGE=(1,), RANGE=(,), and RANGE=(,EQUATEVAL).</p>
HEX64U	accepts a long hex address, in the form <i>nnnnnnnn_nnnnnnnn</i> . The underscore is optional. If an underscore is used, 8 digits must be to the right of it and a maximum of 8 digits may be to the left. If an underscore is not used, a total of 16 digits are allowed. RANGE does not affect this type.
IBMCLASS	accepts a 1-character token representing one privilege class in the ranges A through G. CP converts the token into an 8-bit bit map where the number 1 marks the privilege class that was specified in the token.
INSTRUCT	accepts and converts a 4-, 8-, or 12-digit hexadecimal machine instruction. The instruction must be a valid one as far as its structure and operation code are concerned. The result is a 2-, 4-, or 6-byte value, which is stored in the specified output field.
LNKMODE	accepts one of the following disk access modes: ER, EW, M, MR, MW, R, RR, SR, SM, SW, W, and WR. The DDEVMODE equivalent equate is placed at the specified output location.
NULL	no token is examined. The CP parser uses the NULL conversion type internally.
PASSWORD	accepts a single token not longer than the output field. CP converts the token into uppercase before storing it. After storing it, CP transforms the token so that the location in storage is no longer readily readable. CP then clears the original token using blanks.
PRIVS	accepts a token consisting of one or more privilege classes (characters in the ranges A-Z and 1-6) concatenated together. The result is converted into a 32-bit bit map with a 1 indicating each class specified in the token.

Table 11. Conversion types (continued)

Conversion Type	Meaning
PRODID	accepts and stores a 7 or 8 character token, containing characters only valid in CMS file names, in the output field. Valid characters are: <ul style="list-style-type: none"> • Dollar sign (\$), also called currency symbol • Number sign (#), also called pound sign or hash symbol • Commercial "at" sign (@) • Underscore (_) • Plus sign (+) • Minus sign or dash (–) • Colon (:) • Alphabetic characters (A through Z) • Numeric characters (0 through 9).
REMPRIVS	accepts a token (with no imbedded blanks) consisting of the minus sign (-) followed by one or more privilege classes (alphanumeric characters in the ranges A through Z and 1 through 6). CP converts the token into a 32-bit bit map where the number 1 marks each privilege class that is specified in the token.
REST	accepts the remainder of the line to be parsed and places it in the output field, if the output field is long enough. REST is useful for commands such as MSG.
SCREENSZ	accepts a single token of the form <i>mmmXnnnn</i> , which represents a screen size and stores the result as two fullwords in the specified output field.
SETPRIVS	accepts a token (with no imbedded blanks) consisting of the equal sign (=) followed by one or more privilege classes (alphanumeric characters in the ranges A through Z and 1 through 6). CP converts the token into a 32-bit bit map where the number 1 marks each privilege class that is specified in the token.
SPLCLASS	accepts and converts a single spool file class (A to Z, 0-9).
SPLCLASSES	accepts a collection of spool file classes concatenated together. The collection may be as large as the output field. Each character must be within the ranges A-Z or 0-9.
STRING	a single token or quoted string of a length not to exceed that of the output field. Quotation marks are necessary if you want to maintain imbedded blanks and mixed case in the entered string. If it is specified as a single token without quotation marks, the token is uppercased.
TIMEOFF	accepts a token of the form <i>hh[.mm[.ss]]</i> that expresses a time offset from UTC. The token is converted into the number of seconds that the offset represents and is stored as a fullword in the output field.
TOKEN	a single token not longer than the output field. The token is uppercased before it is stored.
USERID	accepts and stores a 1- to 8-character token, containing characters only valid in user IDs, in the output field. A <i>userid</i> value of a single asterisk (*) will cause the parser to store the user ID from the VMDBK currently addressed by register 11.
USERPTRN	accepts and stores a 1- to 8-character token in the output field, which contain only characters valid in user IDs and the generic characters (* and %).

Examples

Usage notes

1. Storage locations typically are specified in the parser syntax macros by these methods:

```
kw=where
kw=(where,what)
```

except for ACCUM= and NEXT=, which we will ignore. Examples would be:

```
STORE=where
ORFLAG=(where,what)
ANDFLAG=(where,what)
```

"what" is simply anything that would be acceptable in an Or-Immediate (OI) instruction or in an And-Immediate (NI) instruction.

"where" typically is specified in either of these 2 forms:

```
field
base(field)
```

A simple "field" specification is always interpreted as "&PLBASE(field)", so the two forms look the same after this interpretation. Usually, we specify "base" as the name of a DSECT, but this is not necessary. The parser macros simply generate a half-word offset:

```
DC      AL2(field-base)
```

The base address to which this offset is added is the PLBASE= address or the matching BASES= entry as appropriate.

HCPTKDEF saves in assembler variables (for use later by the HCPDOSYN macro) the names of the DSECTs into which data are to be stored. For example, these example keywords:

```
ORFLAG=(SYSCM(SYSIPLFL),SYSAUTOW)
STORE=SYSCM(SYSCKVOL)
STORE=SCFMACH
```

would cause HCPTKDEF to remember the target locations as

```
SYSCM(SYSIPLFL)
SYSCM(SYSCKVOL)
SCFMACH
```

Later, HCPDOSYN will look at each target location. HCPDOSYN first tries to match the target location to something in BASES=. It does this by separating the string into what appears to be a DSECT and a label (in these examples, SYSCM and SYSIPLFL; SYSCM and SYSCKVOL). If it cannot perform the separation, then the target location must be in the primary plist (in this example, SCFMACH must be in the primary plist).

After breaking apart the target location, HCPDOSYN searches for the apparent DSECT name in the BASES= list. If it does not find it, then the target location must be in the primary plist. The original target location string will be used. If it does find it, then it remembers which entry in BASES= satisfied the search. Finally, HCPDOSYN generates the syntax control blocks, where it saves the number 'n' of the plist area ('n' = 0 for the primary plist, 'n' > for something in the BASES= list) and the offset into that DSECT to the target location.

A misspelled "base" will generate a very bizarre error. Consider:

```
HCPTKDEF ...,STORE=SYSCX(SYSIPLFL)
HCPDOSYN ...,PLBASE=MYDSECT,BASES=(SYSCM)
```

The misspelled "SYSCX" means that "SYSCX" will not be found in BASES=. Therefore, it must be in the primary plist, MYDSECT. The offset that the parser will generate will be:

```
DC      AL2(SYSCX(SYSIPLFL) - MYDSECT)
```

This will not be right.

Example 1

Example of the use of 2 separate HCPTKDEF macros to handle a keyword followed by a required variable compared to the use of a single HCPTKDEF macro with both a keyword and TYPE= value specified.

This portion of syntax:

➤ PIECES — *decimal* ➤

could be represented by these parser macros.

```
HCPSTDEF REQMATCH=YES  
HCPTKDEF 'PIECES'  
HCPSTDEF REQMATCH=YES  
HCPTKDEF TYPE=DEC
```

The syntax could be represented more simply by these parser macros, which combine the 2 HCPTKDEF macros into 1.

```
HCPSTDEF REQMATCH=YES  
HCPTKDEF 'PIECES',TYPE=DEC
```

The advantage is a simpler and more compact set of parser macros.

HCPZPRPC — Parse Remainder of Command Line

HCPZPRPC is called to parse a partial command contained in the GSDBK anchored at the VMDCFBUF field in the caller's VMDBK. The syntax description data area passed by the caller describes only the remainder of the command, so processing resumes where it left off (earlier parts of the command may have been processed by the traditional method).

Input Registers:

- | | |
|-----|---|
| R0 | Address of the syntax structure (HCPSYNDF) created by a ROOT=YES invocation of the HCPDOSYN macro. |
| R1 | Address of the primary output area, identified by the PLBASE operand on the HCPDOSYN macro, in which to return information. |
| R2 | Address of the vector of addresses of areas specified on the BASES operand of the HCPDOSYN macro or zero if no BASES were specified. |
| R3 | Address of the command or configuration file statement definition (HCPCFDEF) that defines the operands of the command or statement. If the address is:
<div style="margin-left: 20px;"> 0 (zero)
 tells CP to find the HCPCFDEF.

 non-zero
 tells CP the location of the previously-found HCPCFDEF. </div> |
| R11 | Address of the VMDBK of command invoker (VMDCFBUF → GSDBK, parsing resumes at the offset denoted by GSDSCAN). |

Output Registers — Normal:

- | | |
|-----|------|
| R0 | Zero |
| R15 | Zero |

Output Registers — Error:

- | | |
|-----|--|
| R0 | Error message number as a return code. |
| R15 | <div style="margin-left: 20px;"> 24
 Error encountered paging in the syntax table

 28
 Syntax error encountered in command

 32
 Non-fatal error encountered by post-processor

 36
 Fatal error encountered by post-processor. </div> |

HCPZPRPG — Parse Any GSDBK

HCPZPRPG is called to parse an arbitrary General System Data Block (GSDBK). The syntax description data area passed by the caller describes the syntax of the remaining data in the GSDBK. Parsing continues in the GSDBK at the position indicated by GSDSCAN. Parsing will continue only as far as any X'15' character.

Input Registers:

- R0 Address of the syntax structure (HCPSYNDF) created by a ROOT=YES invocation of the HCPDOSYN macro.
- R1 Address of the primary output area, identified by the PLBASE operand on the HCPDOSYN macro, in which to return information.
- R2 Address of the vector of addresses of areas specified on the BASES operand of the HCPDOSYN macro or zero if no BASES were specified.
- R3 Address of the command or configuration file statement definition (HPCFDEF) that defines the operands of the command or statement. If the address is:
 - 0 (zero)**
tells CP to find the CFDEF.
 - non-zero**
tells CP the location of the previously-found HPCFDEF.
- R4 address of the GSDBK to be parsed.

Output Registers — Normal:

- R0 Zero
- R15 Zero

Output Registers — Error:

- R0 Error message number as a return code.
- R15 **24**
Error encountered paging in the syntax table
- 28**
Syntax error encountered in command
- 32**
Non-fatal error encountered by post-processor
- 36**
Fatal error encountered by post-processor.

Pre-Processing Routines

If you specify a routine on the PREPROC keyword of the HCPDOSYN macro, the parser calls that routine before parsing the command or statement. A pre-processor removes record qualifiers from statements in the system configuration file before it processes the record. No matter if the called routine was dynamically loaded or is part of the CP nucleus, linkage to the called routine is direct linkage and uses a technique known as "call-by-register". For details, refer to ["Discussion of the HCPCALL Macro" on page 18 in Chapter 3, "Creating a Dynamically Loaded Routine," on page 11.](#)

Input Registers Passed to Pre-Processor Routines:

- R0 Address of the CONWK area used by the parser. CONRECBF contains a pointer to the string being processed, CONRECSZ contains the length of the string, and CONBSADR contains a pointer to the vector of addresses mapped by the BASES list on the HCPDOSYN macro. The CONWK area is persistent across statements during the parsing of the configuration file and files that it imbeds.
- R1 Address of the primary output parameter list, identified by PLBASE parameter of the HCPDOSYN macro, provided by the routine that called the parser.

Output Registers — Normal:

- R15 **0**
Statement may be processed
- 4**
Statement should not be processed

Output Registers — Error:

- R15 **8**
Error encountered in processing input string

For more information, see [“General Rules for Coding an Alternate MDLAT Macro”](#) on page 213.

Post-Processing Routines

If you specify a routine on the PROCESS parameter of the HCPCFDEF macro, the parser calls that routine after all processing for the command or statement has completed successfully. No matter if the called routine was dynamically loaded or is part of the CP nucleus, linkage to the called routine is direct linkage and uses a technique known as "call-by-register". For details, refer to [“Discussion of the HCPCALL Macro”](#) on page 18 in Chapter 3, [“Creating a Dynamically Loaded Routine,”](#) on page 11.

Input Registers Passed to Post-Processor Routines:

- R0 Address of the vector of addresses to areas specified on the BASES operand of the HCPDOSYN macro or zero if no BASES were specified.
- R1 Address of the primary output plist provided by routine that called the parser (identified by PLBASE parameter of the HCPDOSYN macro).
- R2 Address of the CONWK area used by the parser. The CONWK area is persistent across statements during the parsing of the configuration file and files that it imbeds.
- R3 Value contained in the CONUWORD field of the CONWK area. This field is set as the result of processing the UWORD= parameter on a HCPTKDEF macro.

Output Registers — Normal:

- R15 **0**
Statement processing completed successfully

Output Registers — Error:

- R0 Message number (including version number) of error message to issue, or zero.
- R1 Parameter list to use when issuing the message (passed to HCPCONSL).

- R15 **4** Error encountered in statement processing
- 8** Fatal error encountered in statement processing.

For more information, see [“General Rules for Coding an Alternate MDLAT Macro” on page 213.](#)

Appendix H. Understanding the CP Message Repository

This appendix describes the format of the CP message repository file and idiosyncrasies related to rules and function of the repository routines.

Message Repository File

The message repository can have any valid CMS file name. The convention for the file name is xxxyyycc, where:

xxx

is a component identifier (for example, HCP).

yyy

usually refers to entry point identifier (for example, MES).

cc

is a 1- or 2-character country code that identifies the language you are working in. Country codes for supported languages are defined in the VMFNLS LANGLIST file. A country code is not used for American English.

The message repository can have any valid CMS file type not reserved for some other function. For a list of reserved CMS file types, see [z/VM: CMS User's Guide](#). The convention is to use a file type of REPOS.

Your message repository must be a fixed-record format file with a maximum logical record length (LRECL) of 80 and should contain the following items:

- Comment records
- A control line
- Message records.

Commenting Your Message Repository

Your message repository file should contain comment records. These must start with an asterisk (*) in column 1:

```
* This is an example of a comment line
```

Comment lines can go anywhere in a message repository file and should describe what is in the file.

Creating a Control Line

The first non-comment record in your message repository must be a control line. The control line contains a single character in column 1 that defines the character you will be using for variable substitutions. For example, the control line of the CP message repository (HCPMES REPOS) is:

```
$
```

This means that the dollar sign (\$) is used throughout the CP message repository to define indicators that will be substituted with data when CP issues messages.

Creating Message Records

Each message record in a CP message repository file contains a maximum of 5 fields. When you create your file using XEDIT, the message records must be in the following format:

```
===== .
===== .
===== .
===== NNNNVVLLS ----- text -----
===== |...+...1...+...2...+...3...+...4...+...5...+...6...+...7...
===== NNNNVVLLS ----- text ----- SID_CODE
===== .
===== .
===== .
```

Figure 14. CP Message Record Format

NNNN

is the message number, in columns 1 through 4. You must use a 4-digit decimal message number in a CP message repository. The CP message repository uses message numbers 0001 through 9999.

Note: In a CP message repository, all message numbers must be in sequential order. CMS (or CMS application) user repositories let you place messages in any order.

VV

is the message version, in columns 5 and 6. The message version is a 2-digit decimal number from 01 through 99 that differentiates one version of a message from any other versions of that same message. If a message only has one version, you can specify blanks in columns 5 and 6 and the version field will default to "01". You cannot use "00" as a version number.

LL

is the line number of the message, in columns 7 and 8. Use this field to define text for a single message that splits into more than one line. You must use a 2-digit decimal line number between 01 and 99. The first line of the message must be 01 and all other line numbers must be sequential and consecutive (that is: 02, 03, 04, and so forth).

If the message has only one line, you do not need to specify a line number, because it defaults to 01 if omitted. You cannot use 00 as a line number.

S

is the severity code, in column 9. Valid severity codes are:

Code

Message Type

E

Error

I

Information

R

Response

A

Immediate action required

D

Decision

W

System Wait

You can define other severity codes by changing the letter in column 9. We recommend that you continue to use the currently defined severities for CP messages so that you do not confuse the users of the system.

text

is the message text, starting in column 11. If your message repository will not contain SID codes (support identification codes), you can specify a maximum of 61 characters of message text per line. If your message repository does or will contain SID codes, you can specify a maximum of 53 characters of message text per line. The next 8 characters are the SID code, which is described below.

If your message text is longer than 61 (or 53) characters, it will not fit on one physical line in the message repository. You must choose whether you want the message to display as:

One line —

specify the identical message number (NNNN), version (VV), and line number (LL) for each line.

Note: If your message text is very long, your message might display as one long continuous line wrapped around to the next physical line of the screen.

When creating a 1-line message (wrapped or not) from a multiple line definition, CP strips all but one of the trailing blanks from the end of the message text and concatenates the lines together. If you want to create a message that displays more than one consecutive blank space:

- If you want more than one blank between two items, split the items onto two lines. Insert as many blanks as you need before the item on the second line.
- If you want more than one blank between words or if you want to maintain alignment of fields, you can use a substitution variable that will have a null or omitted value.

Multiple lines —

specify the identical message number (NNNN) and version (VV), but specify different line numbers (LL) for each line. The first line number must be 01, followed by 02, 03, and so forth.

SID_CODE

is the support identification code, in columns 64 through 71. When you XEDIT a file and specify the SIDCODE *string* option, XEDIT inserts the 8-character *string* in the first eight columns of the last 17 columns (logical record length minus 16) of every line in the file that you add or change. Depending on the string that was chosen, the SID code can help you identify who made the change, and possibly, when and why the change was made.

Message repositories are fixed-record format files with a maximum logical record length (LRECL) of 80. Thus, if you update a message repository with the SIDCODE *string* option, XEDIT inserts the SID code in columns 64 through 71.

Message Repository Idiosyncrasies

The following rules and conventions govern the use of message repositories.

- Message numbers in the range of 7000 through 7999 are treated as responses and are not displayed with a message header.
- All trailing blanks but one are removed from a repository line when the message is generated.
- Trailing blanks in the substitution data are removed when non-column format is used.
- When column format is used:
 - The message in the repository must have a number of blanks following the substitution data symbol, '\$nnn', such that the length of the symbol plus the number of blanks is equal to or greater than the maximum length of the substitution data.
 - If the length of the substitution data is less than the length of the substitution symbol, '\$nnn', blanks will be placed after the substitution data to make it the same length as the '\$nnn'.
- No message text associated with one line number may be longer than 229 characters after the substitution indicators are replaced by the actual substitution data.
- In your module that builds the substitution data, the total length of the substitution data and field separators must be less than 2000 bytes.
- No more than 200 substitution elements may be passed for processing in a single message.

Additional CP Message Repository Idiosyncrasies

Certain message numbers generated for COMPID=HCP cause special processing related to the Protected Application Environment to occur when they are invoked. This processing causes CP to extract data from

the error message substitution data for later use by Diagnose code X'0B0' so the application is protected from seeing the message and entering the CP READ state. These messages include message numbers: 410, 450, 452, 453, 650, 657, 1459, 9300, and 9501. It is recommended that installations refrain from adding or changing versions of these messages.

Appendix I. Updating the CP Load List

This appendix describes the CP load list and how you can update it to include your dynamically loaded modules. The CP load list is an ordered listing of modules in the CP nucleus. The HCPMDLAT (module attribute) macroinstruction defines the location of a module in the CP nucleus and the linkage attributes of that module.

Traditionally, when you customized your z/VM system by adding new CP modules, you provided source updates to HCPMDLAT that would include your routines as part of the CP nucleus. You can completely avoid modifications to the CP load list by coding your routines so that they are loaded dynamically. This is the preferred technique from the standpoint of managing your local modifications.

However, there may be cases when you might still want to update the CP load list. For example, as part of an existing local modification, you may have a routine which you have not had a chance to modify so that it can be loaded dynamically. Or, perhaps some design consideration requires that you still place certain routines within the CP nucleus. In those instances, it is still possible to avoid source modifications to the IBM supplied HCPMDLAT MACRO. This is accomplished by using an alternate MDLAT MACRO. You will often see this referred to as an xxxMDLAT MACRO. The xxx represents an alternate component ID, that is, a component ID that you have selected for your routines.

The following is the basic steps that you need to do in order to add a user module to the CP nucleus using an Alternate MDLAT macro.

1. Create the alternate MDLAT macro and place it in a maclib where GENCPBLS will find it.
2. Code the user module.
3. Use GENCPBLS to generate the new CP load list.
4. Use VMFBLD to generate the CP nucleus.

General Rules for Coding an Alternate MDLAT Macro

You use 3 macros (MDLATHDR, MDLATENT, MDLATTRL) to create an alternate MDLAT macro. The alternate MDLAT macro contains a single header macro (MDLATHDR) at the beginning and it contains a single trailer macro (MDLATTRL) at the end. In between these, code as many MDLATENT macros as you require to define your modules. The syntax for these macros is described in [Appendix F, "CP Exit Macros,"](#) on page 169.

The alternate MDLAT macro can be used to identify the location of your module in the CP nucleus. Each section of the CP nucleus is identified by a memory marker that has the generic format HCPMMn. For example, the memory marker for the start of the resident MP section of the CP nucleus is HCPMM1. Your modules are placed in the right section of the load list based on the HCPMMn memory marker that they follow in your alternate MDLAT MACRO. For information about the different sections of the CP nucleus and the name of the associated memory marker, refer to the documentation contained in the HCPMDLAT MACRO.

The alternate MDLAT macro also identifies the linkage attributes for your module. This information is used by the standard CP linkage macros (for example, HCPCALL or HCPENTER) in order to generate the correct assembler instructions to accomplish the linkage.

When you create an alternate MDLAT macro, keep the following general rules in mind:

- Your modules are placed in the right section of the load list based on the HCPMMn memory marker that they follow in your alternate MDLAT MACRO. The exact placement within that section is not predictable.
- There is no default section in the CP load list where modules will be added. That means you must specify at least one HCPMMn memory marker if your alternate MDLAT macro defines any modules that you expect to be placed in the CP load list.

- If your module will be dynamically loaded, you specify CPXLOAD=YES on the MDLATENT macro so that it will not be added to the CP load list. If all your module entries in the alternate MDLAT macro are defined as dynamically loaded, then you do not have to specify any HCPMMn memory marker.
- It is not necessary to specify all the memory markers. Only use the ones that you need in order to specify the section of the load list in which you wish to place your modules.
- The order of the HCPMMn entries in your alternate MDLAT MACRO is not significant. Your modules will be placed in the right section of the load list based on the HCPMMn memory marker that they follow in your alternate MDLAT MACRO.
- Multiple occurrences of a specific HCPMMn memory marker are allowed.

Creating an Alternate MDLAT Macro

Use the XEDIT command to create an xxxMDLAT MACRO file. The format of the file must conform to the requirements of a program written in assembler macro language as defined in *HLASM MVS & VM Language Reference*.

Figure 1 shows an example of an alternate MDLAT macro called ABCMDLAT MACRO. This alternate MDLAT macro defines five new modules, four of which are to be included in the CP nucleus and one which is expected to be loaded dynamically.

```
+-----+
| ABCMDLAT MACRO      A1 F 80 Trunc=72 Size=23 Line=8 Col=1 Alt=1
|
| |...+....1....+....2....+....3....+....4....+....5....+....6....+....7.>
| 0 * * * Top of File * * *
| 1      MACRO
| 2 &LABEL  ABCMDLAT &EPNAME
| 3      .*
| 4 &LABEL  MDLATHDR &EPNAME
| 5      .*
| 6          MDLATENT HCPMM1,MODATTR=(MP,DYN)
| 7          MDLATENT ABCTST,MODATTR=(MP,DYN),
| 8              EP=((ABCTSTEP,STAT))
| 9      .*
|10          MDLATENT HCPMM5,MODATTR=(MP,DYN)
|11          MDLATENT ABCMOD,MODATTR=(MP,DYN)
|12      .*
|13          MDLATENT HCPMM1,MODATTR=(MP,DYN)
|14          MDLATENT ABCKLM,MODATTR=(MP,DYN),
|15              EP=((ABCKLMST,STAT))
|16          MDLATENT ABCIJK,MODATTR=(MP,DYN)
|17      .*
|18          MDLATENT ABCCMD,MODATTR=(MP,DYN),CPXLOAD=YES
|19          MDLATENT ABCSRV,MODATTR=(MP,DYN,TMAR),
|20              EP=((ABCSRVXX,AM64E,LREGE)),CPXLOAD=YES
|21      .*
|22          MDLATTLR
|23      .*
|24          MEXIT
|25          MEND
|26 * * * End of File * * *
```

Figure 15. ABCMDLAT Macro

A brief explanation of the significant lines in this alternate MDLAT MACRO file follows.

- The following statements should be coded exactly as shown:
 - Line 1 - Macro definition header statement
 - Line 4 - MDLATHDR macro

This macro does some actions necessary to initialize for processing of the subsequent MDLATENT macros.
 - Line 22 - MDLATTLR macro

This macro does some actions necessary to complete the definition of the alternate MDLAT macro.

- Line 24 - MEXIT instruction
- Line 25 - Macro definition trailer statement
- Lines 3, 5, 9, 12, 17, 21, and 23

These are internal macro comment statements. They are used to improve the readability of the macro.

- Line 2

This is the macro instruction prototype statement. This statement assigns the name ABCMDLAT to the macro. It also declares &EPNAME as a parameter. This parameter is passed without modification to the MDLATHDR macro.

- Lines 6, 7 and 8

These two MDLATENT macros will add your new module ABCTST to the MP nucleus section that starts with the HCPMM1 memory marker. All the entry points in ABCTST will use CP dynamic linkage except for ABCTSTEP, which will use CP static linkage.

- Lines 10 and 11

These two MDLATENT macros will add your new module ABCMOD to the non-MP nucleus section that starts with the HCPMM5 memory marker. All the entry points in ABCMOD will use CP dynamic linkage.

- Lines 13 through 16

These three MDLATENT macros define the user modules ABCKLM and ABCIJK to the MP nucleus section that starts with the HCPMM1 memory marker. These lines illustrate several points. You are allowed to have multiple occurrences of the same HCPMM n memory marker. In addition, you do not have to ensure that these memory markers are in any numeric order. Also, you can have multiple modules defined after one HCPMM n memory marker.

- Lines 18

This MDLATENT macro defines the module ABCCMD, which will not be added to the CP load list. The CPXLOAD=YES operand indicates that the CP CPXLOAD command or configuration file statement will be used to load the routine dynamically so CP can use it.

- Lines 19 and 20

This MDLATENT macro defines the module ABCSRV, which will not be added to the CP load list. The CPXLOAD=YES operand indicates that the CP CPXLOAD command or configuration file statement will be used to load the routine dynamically so CP can use it. The module will always execute in Access Register mode. The module will always be entered in 32-bit addressing mode, except for entry point ABCSRVXX, which will be entered in 64-bit addressing mode.

- Lines 18 through 20

The scenario we envision here is that ABCCMD is going to handle some new CP command. Also, we plan to load ABCCMD and ABCSRV with CPXLOAD so that they can call each other directly. When the new CP command is executed, ABCCMD will be called TYPE=INDIRECT (because it was dynamically loaded), so its entry point must be DYN AM31 SREG. ABCCMD may, however, call ABCSRV entry points even though they are defined with broader attributes. These calls must be TYPE=DIRECT, and they could even be deferred (by using HCPSTK).

Coding the User Module

The first thing that you will need to do is decide the name of the user module. Refer to “How Should the Routine be Named?” on page 30 and follow its advice. For example, let's suppose you choose the name 'ABCMOD' and it has an entry point called 'ABCMODEP'.

When you write the code for your module, be sure to include an invocation of the HPCMPID macro in order to define the component ID of your module. The component ID will be used during assembly of your module to resolve the system linkage to your entry points in your module. It tells the assembler which alternate MDLAT macros to look in to find the attributes of your module. In our example, you would code a 'HPCMPID COMPID=(ABC)' in the beginning of your module.

You also need to code a HCPCMPID macro in the module that you updated to call your module. For example, if you updated HCPNOS to call entry point ABCMODEP in your module, you will need to code a 'HCPCMPID COMPID=(ABC)' in HCPNOS so that HCPCALL will generate the correct linkage.

Generating a New CP Load List

Once you have created your alternate MDLAT file and specified each module that you want to define in the CP nucleus you use the VMSES/E command, GENCPBLS, to create the CP loadlist. For more information on the GENCPBLS command, see [z/VM: VMSES/E Introduction and Reference](#). There is also a section in [z/VM: Service Guide](#) that shows how to use the GENCPBLS command.

It is important to note that your alternate MDLAT macro (xxxMDLAT) must be in a maclib specified on the MACS statement of the control file that is used by the GENCPBLS command in order to be found. You may update an existing maclib to accomplish this. You may also create your own maclib and update the control file so that it includes your maclib name.

If your alternate MDLAT contains only modules that will be dynamically loaded, then there is no need to generate a new CP load list.

Appendix J. Samples of Dynamically Loaded Routines

This appendix discusses the sample CP exit routines that are shipped on the CP tools disk (usually MAINT 193). We are providing these sample CP exit routines for the following reasons:

You can try out the CP Exit support without first having to write your own dynamically loaded CP routines.

The functions provided by these samples are functions that most users of the CP Exit support would find useful.

When you do write your own dynamically loaded CP routines, you have samples of typical code to use as a pattern. For example, if you needed to use a component ID block (CMPBK), you could see how the samples perform that task and modify the code to meet your installation's needs.

Important Note

The CP exit routines mentioned in this appendix are to be used only as samples. Although the samples may have been reviewed by IBM for accuracy in a specific environment, there is no guarantee that the same or similar results will be obtained elsewhere. The sample CP exit routines are being provided on an "as is" basis without any warranty expressed or implied.

The samples shipped on the CP tools disk are:

1. **XDISPL SAMPASM** — a sample exit routine that can be used for debugging. The entry points provided in this sample are designed to display CP exit point entry information (CP exit number, register contents, parameter list entries, and so forth).

You can use one of the routines in this module to display "before and after" values. For example, suppose you wrote a dynamically loaded CP routine, XYZ. You could issue this ASSOCIATE EXIT command (or configuration file statement):

```
associate exit 1200 enable epname xdispldt xyz xdispldt
```

Then, when CP Exit 1200 is called, you will see the parameter list as it was initialized by HCPDIA and after it was changed by your entry point XYZ. This can be helpful when you are trying to understand what XYZ is doing.

2. **VREPOS SAMPASM** — a sample command handler that takes the messages or responses that you specify and displays them at the console with the specified substitution data. That is, you can use this routine to verify the format of any message in a message repository. So, you could dynamically define a command (for example, CPXMITMSG) and then use that new command to verify the format of a message in the CP message repository (HCPMES REPOS) or in one of your local message repositories.
3. **JAFEXS SAMPASM** — a sample that shows the syntax definition and post-processor routine for an EXTERNAL_SYNTAX example.
4. **JAMSAMA SAMPASM** — a sample routine for CP Exit 1200.
5. **JAMSAMB SAMPASM** — another (more complex) sample routine for CP Exit 1200.

The rest of this appendix is devoted to listing and explaining the two sample CP exit routines for CP Exit 1200, which allow you to examine and, optionally, change or reject the operands specified on the CP DIAL command. These samples are practical examples that you can use as a basis for exploiting CP exit routines on your z/VM system.

For each sample, there is an assembler language listing with comments discussing certain statements of interest. In general, the assembler language listings do not show macroinstruction expansions or copy files. However, there are a few interesting exceptions.

The second example is more complex than the first sample. So, in addition to the annotated assembler language listing, there is also a local message repository (with comments) and a control data file.

What You Should Gain from the Sample CP Exit 1200 Routines

After reviewing (and perhaps even using) these two sample CP exit routines, we hope that you:

- Recognize how powerful the CP Exits support is
- Feel comfortable using the CP Exits support
- Realize that CP exit routines are easy to write
- Have acquired a few new skills, such as:
 - Manipulating a component identification block (CMPBK) — your own private extension to the system common area (SYSCM)
 - Reading a CMS file
 - Calling a CP exit point
 - Controlling a CP exit point
 - Using the CP parser
 - Using your own local message repository.

Understanding CP Exit 1200

CP Exit 1200 allows you to examine and, optionally, change or reject any information provided for the CP DIAL command. This information includes the:

Command entered by the user —

This would be 'DIAL', unless your system has defined a new command or an alias that also called the HCPDIAL module. Whatever the user entered (command or alias) is passed on to the CP exit routine.

Target user ID —

For example, suppose the user entered 'DIAL PVM', but you would rather have the target user ID virtual machine be 'PVMTEST'. Using CP Exit 1200, you can change the target user ID from 'PVM' to 'PVMTEST'.

If the user did not specify a target user ID, your CP exit routine can supply it. If the user did not specify a target user ID and your CP exit routine does not specify one, the command will fail because DIAL needs to connect to a target user ID.

Virtual device number of the target user ID —

For example, suppose the user entered 'DIAL PVM 120', but you would rather have the target virtual device be '220'. Using CP Exit 1200, you can change the target virtual device from '120' to '220'.

Or, suppose you wanted to delete the target virtual device and pretend that none was specified. Your CP exit routine can do this by changing the target virtual device number from '120' to '-1', which is a special value that indicates that no specific virtual device number was specified.

If the user did not specify a target virtual device, your CP exit routine can supply it. If the user did not specify a target virtual device number and your CP exit routine does not specify one, CP will search for an available virtual device number to use, within some range of virtual device numbers.

Range of virtual device numbers —

If the target virtual device number ends up being '-1' (either because the user did not specify it or because CP Exit 1200 changed it), CP will restrict the search for an available virtual device to the specified range. The default range is X'0000' through X'FFFF'.

If the target virtual device number was specifically determined (either because the user specified it or because CP Exit 1200 changed it), CP will ignore the specified range.

Understanding the Sample CP Exit 1200 Routines

These sample CP exit routines allow you to customize the processing of the CP DIAL command. To customize the DIAL processing, these samples actually customize the control data supplied by the user

(command name, target user ID, target virtual device number or range). To accomplish this, the samples use a table that associates the user-specified command name with the information to be returned to the DIAL processor.

Sample 1

The first sample CP exit routine (page [Table 12 on page 223](#)) defines the table entries in a control section (CSECT) during assembly. Specifically, Sample 1 can be described as a table lookup subroutine:

- Validate the input (specifically, the CP exit number)
- Search the table for the incoming command name or target user ID
 - If found, return these values:
 - Target user ID
 - Target virtual device number
 - Start of virtual device range
 - End of virtual device range
- Exit.

Sample 2

The second sample CP exit routine (page [Table 13 on page 242](#)) defines the table in a CMS file and reads that file into the system execution space. Specifically, Sample 2 can be described as a more complicated table lookup subroutine than Sample 1:

- Validate the input (specifically, the CP exit number)
- Check to see if CP Exit F200 (our local CP exit routine) is enabled
 - If enabled, then call it to load the CMS file into the system execution space
- Locate our component ID block (CMPBK)
- Acquire a shared lock on the CMPBK
- Search the table for the incoming command name or target user ID
 - If found, return these values:
 - Target user ID
 - Target virtual device number
 - Start of virtual device range
 - End of virtual device range
- Relinquish the CMPBK lock
- Exit.

The second sample calls local CP Exit F200, which loads a CMS source file into the system execution space:

- Validate the input (specifically, the CP exit number)
- Locate the component ID block (CMPBK) for component ID 'JAM'
 - If found, then acquire a shared lock on the CMPBK
 - If not found, then:
 - Allocate a new CMPBK
 - Acquire a shared lock on the new CMPBK
- Check to see if CP Exit F200 is disabled
 - If disabled, then exit

Sample Routines

- Disable CP Exit F200
- Get a CP data request block (DRBK) for reading the table in the CMS file
- Get a general system data block (GSDBK) for parsing
- Interconnect the DRBK and the GSDBK
- Open the DRBK
 - If the open fails, then exit
 - If the open succeeds, then do until end of file:
 - Read the next record
 - If the record is sparse, all blanks, or a comment, then go back to reading the next record
 - Get a new JAMTABLE
 - Call the CP parser
 - If the call fails:
 - Generate an error message
 - Go back to reading the next record
 - Add the JAMTABLE to the new chain
 - If EOF, end
- Close the file
- Release the DRBK, GSDBK, and the unused JAMTABLE
 - If we had no errors, then:
 - Swap the new chain of JAMTABLEs for the old chain
 - Release the old JAMTABLEs
 - If we had errors, then release the new JAMTABLEs
- Exit.

Why Sample 2 Is Better Than Sample 1

Sample 1 is simple and straightforward to understand, but because the table is created during assembly, it is also rigid and difficult to change after assembly.

Sample 2 is more complicated, but much more flexible. You can reload the table at any time using a simple system operator command:

```
enable exit f200
```

Without local CP Exit F200, Sample 2 would need a way to indicate that the table needed to be reloaded. To do this, the alternatives would be to create a new:

- CP command
- User Diagnose code

Using the Sample CP Exit 1200 Routines

If you decide to use one of the sample CP exit routines listed in this appendix, you must load the files into the system execution space, assign one or more entry points, and enable CP Exit 1200. To accomplish this, you would enter the following commands:

To Use Sample 1:

1. Load the sample (JAMSAM) into the system execution space using the following CP command or system configuration file statement:

```
cpxload jamsam text temporary nocontrol
```

2. Assign entry point JAMSAMD1 to CP Exit 1200 and enable CP Exit 1200 using the following CP command or system configuration file statement:

```
associate exit 1200 enable epname jamsamd1
```

See [Table 12 on page 223](#).

To Use Sample 2:

1. Load the sample (JAMSAM) into the system execution space using the following CP command or system configuration file statement:

```
cpxload jamsam text temporary nocontrol
```

2. Assign entry point JAMSAMD1 to CP Exit 1200 and enable CP Exit 1200 using the following CP command or system configuration file statement:

```
associate exit 1200 enable epname jamsamd1
```

3. Assign entry point JAMSAMD1 to CP Exit F200 and enable CP Exit F200 using the following CP command or system configuration file statement:

```
associate exit f200 enable epname jamsamd1
```

See [Table 13 on page 242](#), [Table 14 on page 289](#), and [“A Sample JAMTABLE SOURCE File” on page 291](#).

Potential Improvements

As an exercise to get used to creating your own CP exit routines, you could make improvements to the second sample CP exit routine in this appendix. For example, you might want to add code to Sample 2 that would:

- Notify the system operator when CP Exit F200 succeeds
- Give the system operator a heading regarding the error messages
- Call search routines using BAS rather than HCPLCALL
- Change the structure of JAMTABLE. Currently, you cannot change the structure of JAMTABLE without careful and judicious use of the following:
 - Creating a new module
 - Using the CPXLOAD statement
 - Using the ASSOCIATE EXIT statement
 - And so forth

Table 12. Annotated Listing for Sample CP Exit Routine 1 for CP Exit 1200

Assembler Source		Commentary
12	MACRO	Line 12: This macro, JAMMDLAT, describes the modules and entry points for component ID JAM. This macro can be included inline, as shown here, or in a MACLIB that is processed by the compiler. The HPCMPID macro, invoked on line 19, adds JAM to the component IDs for this compile.
13	JAMMDLAT &EPNAME	
14	MDLATHDR &EPNAME	
15	MDLATENT JAMSAM,MODATTR=(MP,DYN),	
	CPXLOAD=YES	
16	MDLATTLR	
17	MEND	
		012000001 013000001 014000001 X015000001 016000001 017000001 018000001

Table 12. Annotated Listing for Sample CP Exit Routine 1 for CP Exit 1200 (continued)

Assembler Source		Commentary
19 20	HPCMPID COMPID=JAM COPY HCPOPTMS	<p>Line 19: HPCMPID is a macroinstruction whose purpose is to specify your component ID entries. CP uses the component ID entries to identify your xxxMDLAT macroinstructions, which CP uses to assign linkage attributes. Component ID entries are 3-character strings. For example, IBM's component ID for CP is 'HCP'.</p> <p>Component ID entries can also be used on HPCONSL MACROs, which would connect the HPCONSL MACRO with associated message repositories (see the CP ASSOCIATE MESSAGE command for more information). HPCMPID may be specified multiple times, and with multiple component ID entries. For example:</p> <div>HPCMPID COMPID=(JAM,JAM,XYZ) HPCMPID COMPID=IJK</div> <p>The HPCMPID MACRO is tied to the support for multiple HCPMDLAT MACROs, each known as xxxMDLAT. The 'xxx' letters represent the component ID entries specified in your HPCMPID MACROs. Based on the HPCMPID MACROs shown above, the xxxMDLAT MACROs that HPCALL would use are:</p> <div>HCPMDLAT MACRO always checked first JAMMDLAT MACRO JAMMDLAT MACRO XYZMDLAT MACRO IJKMDLAT MACRO</div> <p>JAMMDLAT is shown twice because JAM is specified twice in the HPCMPID MACROs.</p>

Table 12. Annotated Listing for Sample CP Exit Routine 1 for CP Exit 1200 (continued)

Assembler Source	Commentary
	<p>Line 19 (continued): HPCALL and other MACROs call your xxxMDLAT MACRO looking for the attributes of the module in order to know the proper linkage to generate. HPCALL will call your xxxMDLAT MACROs in the order that your component ID entries are specified on the HPCMPID MACROs.</p> <p>The format of your xxxMDLAT MACROs is similar to the format of the HCPMDLAT MACRO. Here is our JAMMDLAT MACRO, also shown in the listing at line 12:</p> <pre>MACRO JAMMDLAT &EPNAME MDLATHDR &EPNAME MDLATENT JANSAM,MODATTR=(MP,DYN), X CPXLOAD=YES MDLATTLR MEND</pre> <p>You use the MDLATENT MACRO to specify the attributes of your module using the same keywords as you used in HCPMDLAT. For a typical module, what you see here is all that you would need. Only if your module has special attributes beyond these would you investigate other MDLATENT keywords.</p> <p>Attribute: Meaning:</p> <p>MP - Multiprocessor capable. This means that tasks on different processors may run this module simultaneously.</p> <p>DYN - Uses a dynamic savearea (SAVBK).</p> <p>Attributes, specifically MP or NONMP, specified here must match the attributes specified by CPXLOAD.</p>

Table 12. Annotated Listing for Sample CP Exit Routine 1 for CP Exit 1200 (continued)

Assembler Source		Commentary	
000000 000000 000000 000002 000004 000008 000000	347 JAMSAM	HCPPROLG ATTR=(RESIDENT,REENTERABLE), COPYR=(1995,2005),COPYRID='My Copyright', BASE=(R12)	X02300001 X02400001 02600001
	480+JAMSAM	CSECT ,	
	481+HCP@MOD	DS ,	
	482+ DC AL2(C'V')	Bogus instruction to prevent fall-thru	&VXIN0ZW 01-HCPRR
	483+ DC C'VV'	For HCPENTER enforcement characters	@VRGB1QY 01-HCPRR
	484+ DC A(0)	No call- or goto-by-register vector	@VRGB1QY 01-HCPRR
	485+ DC A(0)	For PLX, branch around prologue	@VRGB1QY 01-HCPRR
	+	For ASM, spacer	@VRGB1QY
00000C 000016 00001E 000026	918194A28194A040		
	C5D4C5F2F0F50000		
	F0F761F2F961F0F5		
	7AF1F04BF4F5D9C5		
00002E 00003A 00003E 00003F 000048	D4A840C39697A899		
	F1F9F9F5		
	6B		
	F2F0F0F4		

Line **347**: The keyword 'COPYRID=' is used only if the keyword 'COPYR=' is specified. The string assigned to 'COPYRID=' will be assembled as a character constant in place of the IBM copyright notice.

Lines **482-485**: Here you see "bogus instruction", enforcement characters, address of vector, and so on. These fields mirror fields generated by HCPENTER to support call-by-register and goto-by-register. None of these data fields are executable instructions, so execution cannot fall through into any HCPENTER macro, which is why there is a "bogus instruction".

Line **486**: The module name is saved in lower case whenever it does not start with 'HCP'. Services in CP may use this value as stored here (TRACE instructions for HPCALL/HCPEXIT linkage) or may convert it to upper case (HCPCONSL when building error message headers). This is just another peculiarity that you should be aware of.

Lines **490-493**: You can use the 'COPYR=' **and** 'COPYRID=' keywords to insert your own copyright information into the expansion of the HCPPROLG macro. Notice that both keywords must be specified, or else your copyright information will not be generated.

Table 12. Annotated Listing for Sample CP Exit Routine 1 for CP Exit 1200 (continued)

Assembler Source			Commentary
000048 000060	000048 00218	571+HCPDATAA LOCTR ,	Lines 571-573 : LOCTR instructions are used to place data early in the module (in this example, data starts at offset 0x000048) followed by executable code (in this example, code starts at offset 0x000060). Certain macros will generate LOCTR instructions in order to add their data to these sections. So you may find instructions that are sequential in the listing but are not sequential in memory. There will be more examples of this in Sample Exit Routine 2.
	000060 00218	573+HCPCODEA LOCTR ,	
		@Y0656QY 02 - HCPDA @Y0656QY 02 - HCPDA	
	575	PRINT ON,NOGEN	02700001
	577	COPY HCPEQUAT - General equates	
	3455	COPY HCPEQXIT - Equates for Exit control	03000001
Line 3455 : HCPEQXIT is a good place to look for exit equates. Exit equates should all be of the form: XIT@xxxx EQU X'xxxx'. As always, we suggest that you do not add your exit number equates to this IBM COPY file. Add them to your own COPY file instead. IBM-defined CP exit routines should have equates of this form in the HCPEQXIT COPY file. The intent is that every exit point should be listed here, so that we can tell what has been used, with a small amount of information about what it is for. The two exits in DIAL processing are assigned these CP exit numbers:			
<div>XIT@1200 EQU X'1200' Command: DIAL * Validate operands passed * on the DIAL command * XIT@1201 EQU X'1201' Command: DIAL * Second chance, after * target has been decided. *</div>			

Table 12. Annotated Listing for Sample CP Exit Routine 1 for CP Exit 1200 (continued)

Assembler Source		Commentary	
3616		COPY HCPPFXPG - Host Prefix Page	03100001
000000	6589	X1200 * * X1200UWD DS X1200TRT DS X1200CMD DS X1200UWD DS X1200VDS DS X1200VDS DS X1200VDE DS X1200RDV DS 6975 7342 11605 11608 11611	03200001 70500210 71000210 71500210 72000210 72500210 73000210 73500210 74000210 74500210 75000210 75500210 76000210 03300001 03400001 03600001 03700001 03800001
		HCPPLXIT - PLIST definitions for IBM Exit Points The PLIST pointers for exit 1200 Read all comments here as if they all begin "Address of" 1 Reserved 2 User words 3 TRT 256 bytes 4 Command name 5 Target userid 6 Target VDEV number 7 VDEV number start search range 8 VDEV number end search range 9 RDEV HCPSAVBK - Savearea Block HCPVIMBK - Virtual Machine Definition Block HCPUSING PFPG,R0 HCPUSING VMDBK,R11 HCPUSING SAVBK,R13	

Line **6589**: HCPPLXIT is the place to look for plist definitions for IBM-defined CP exit points. As always, we suggest that you do not add your exit plist definitions to this IBM COPY file. Add them to your own COPY file instead. Every exit plist should be of the form:

```
Xxxxx DSECT /
XxxxxUWD DS A Reserved
... words doubleword aligned ... initialized to binary ... zeros
XxxxxTRT DS A Address of 256 bytes
... doubleword aligned ... initialized to binary
Xxxxx... DS A ... zeros
Xxxxx... DS A Address of a data item
Xxxxx... DS A Address of a data item
```

Storage for the plist is allocated in the mainline routine for each instance of calling an exit. The plist and what it points to are untouched by the exit control routines. Each exit routine sees whatever the prior exit routine left. The storage is deleted upon final return from all of the exit routines.

Plist addresses after the first three standard entries all depend on the exit. In fact, the exit need not conform to using the first three standard entries. How the mainline builds the plist for its exit may be rather arbitrary.

Table 12. Annotated Listing for Sample CP Exit Routine 1 for CP Exit 1200 (continued)

Assembler Source		Commentary	
0000000	000000 00020	11615 JAMTABLE DSECT , 11616 JAMTCMD DS CL12 11617 JAMTUID DS CL(L'VMDUSER) 11618 JAMTVDN DS F 11619 JAMTVDS DS F 11620 JAMTVDE DS F 11621 JAMTABLN EQU *-JAMTABLE	My command-to-userid table Command Target userid Target VDEV number VDEV number range start VDEV number range end
000000C			04000001
0000014			04100001
0000018			04200001
000001C	00020		04300001
			04400001
			04500001
			04600001
Line 11615: This is our local DSECT. It maps the storage defined later at line 17877.			
11623 *	* Start of specifications *****		
11624 *	* Entry Point Name - JAMSAMD		
11626 *	* Descriptive name -		
11628 *	* Dial exit 1200		
11629 *	* Function -		
11630 *	* Preprocess DIAL command input		
11632 *	* Register usage -		
11633 *	* R0 - Exit number		
11634 *	* R1 - Address of exit plist pointers		
11635 *	* R2 -		
11636 *	* R3 -		
11637 *	* R4 -		
11638 *	* R5 -		
11639 *	* R6 -		
11640 *	* R7 - Address of JAMTABLE		
11641 *	* R8 -		
11642 *	* R9 - Address of exit 1200 plist		
11643 *	* R10 -		
11644 *	* R11 - Dispatched VMDBK address		
11645 *	* R12 - Base register		
11646 *	* R13 - Save Area address		
11647 *	* R14 - Work register, linkage		
11648 *	* R15 - Work register, linkage		
11649 *	* SAVEMRK usage -		
11650 *	* SAVEMRK0 -		
11651 *	* SAVEMRK1 -		
11652 *	* SAVEMRK2 -		
11653 *	* SAVEMRK3 -		
11654 *	* SAVEMRK4 -		
11655 *	* SAVEMRK5 -		
11656 *	* SAVEMRK6 -		
11657 *	* SAVEMRK7 -		
11658 *	* SAVEMRK8 -		
11659 *	* SAVEMRK9 -		
11660 *	* SAVEMRK10 -		
11661 *	* SAVEMRK11 -		
11662 *	* SAVEMRK12 -		
			04800001
			04900001
			05000001
			05100001
			05200001
			05300001
			05400001
			05500001
			05600001
			05700001
			05800001
			05900001
			06000001
			06100001
			06200001
			06300001
			06400001
			06500001
			06600001
			06700001
			06800001
			06900001
			07000001
			07100001
			07200001
			07300001
			07400001
			07500001
			07600001
			07700001
			07800001
			07900001
			08000001
			08100001
			08200001
			08300001
			08400001
			08500001
			08600001
			08700001

Table 12. Annotated Listing for Sample CP Exit Routine 1 for CP Exit 1200 (continued)

Assembler Source	Commentary
11663 * Input - 11664 * R0 - Exit number 11665 * R1 - Address of plist addresses 11666 * R2 - Address of XCRBK 11667 * Output - See Exit normal, Exit error 11668 * 11669 * Exit normal - 11670 * R15 = 0 11671 * Target userid changed 11672 * 11673 * Exit error - 11674 * None 11675 * 11676 * General comments - 11677 * The original plist built for exit 1200 was this: 11678 * 11679 *	<p>Line 11663: The standard contents of the registers at entry to all exit routines will be like this:</p> <ul style="list-style-type: none">• R0 will contain the exit number.• R1 will contain the address of an array of addresses (the plist). The high order bit of the last address will be on.• R2 will contain the address of an XCRBK (Exit Call Request Block).• R3-R10 will be garbage.• R11 may contain the address of the dispatched VMDBK, or maybe not. That all depends on the exit.• R12 will be used as the module base register.• R13 will be used as the SAVBK base register.• R14-R15 will be used as linkage registers. <p>Storage for the XCRBK is allocated or deleted by ASSOCIATE EXIT processing. An XCRBK is allocated for each epname specified by ASSOCIATE EXIT. If an epname is specified more than once, each occurrence gets a separate XCRBK. The user words (Reserved for non-IBM use) are initialized to binary zeros when allocated but are otherwise untouched by the exit control routines.</p>

Table 12. Annotated Listing for Sample CP Exit Routine 1 for CP Exit 1200 (continued)

Assembler Source

Commentary

Line **111663** (continued): In this example, we do not use HCPXCRBK, so we do not show it in the listing. Here is what it looks like:

XCRBK	DSECT	//// (various control fields)
* XCRUSD1 DS	D	Reserved for non-IBM use
XCRUSD2 DS	D	Reserved for non-IBM use
XCRUSRF1 DS	F	Reserved for non-IBM use
XCRUSRF2 DS	F	Reserved for non-IBM use
XCRUSRH1 DS	H	Reserved for non-IBM use
XCRUSRH2 DS	H	Reserved for non-IBM use
XCRUSRX1 DS	X	Reserved for non-IBM use
XCRUSRX2 DS	X	Reserved for non-IBM use
XCRUSRX3 DS	X	Reserved for non-IBM use
XCRUSRX4 DS	X	Reserved for non-IBM use
* XCRFWD DS	A	Address of next XCRBK
* XCRATMPT DS	F	Count of attempts to call this routine
* XCRCALLS DS	F	Count of calls completed
* XCRMSACT DS	D	Reserved
* XCR\$END DS	00	Time (in micro-seconds) that ... this routine was active. The end
*		

Table 12. Annotated Listing for Sample CP Exit Routine 1 for CP Exit 1200 (continued)

Assembler Source		Commentary
11680 * 11681 * 11682 * 11683 * 11684 * 11685 * 11686 * 11687 * 11688 * 11689 * 11690 * 11691 * 11692 * 11693 * 11694 * 11695 * 11696 * 11697 * 11698 * 11699 * 11700 * 11701 * 11702 * 11703 * 11704 * 11705 * 11706 * 11707 * 11708 * 11709 * 11710 * 11711 * 11712 * 11714 * 12300 * 12301 * 12302 * 12303 * 12305 * 12333 * 12334 * 12335 *	<pre>PLIST=('RESERVED', Argument 1 'USERWORD', Argument 2 'TRTTABLE', Argument 3 SAVCMD, Argument 4 DIALEUSR, Argument 5 VNUMBIN, Argument 6 VNUMSTRT, Argument 7 VNUMEND, Argument 8 RDEV) Our purpose is to customize the DIAL command processing. Given some control data (either the command name that caused the DIAL processing to be driven or the target userid), we will customize the DIAL processing. To do so, we need a table that associates the incoming command name with the data to return to DIAL. We will define the table entries in this CSECT during assembly. Our logic: - validate input (specifically, exit number) - search the table for the command or target userid - if found, then return these - target userid - target vdev - vdev range start - vdev range end - exit ***** End of specifications *****</pre>	Line 11680: The following comments are copied from HCPDIA and are here for you to edit as needed:
10500001 10600001 10700001 10800001 10900001 11000001 11100001 11200001 11300001 11400001 11500001 11600001 11700001 11800001 11900001 12000001 12100001 12200001 12300001 12400001 12500001 12600001 12700001 12800001 12900001 13000001 13100001 13200001 13300001 13400001 13500001 13600001 13700001 13900001 14100001 14200001 14300001 14400001 14600001 17400001 17500001 17600001	<pre>PLIST=('RESERVED', Argument 1 'USERWORD', Argument 2 'TRTTABLE', Argument 3 SAVCMD, Argument 4 DIALEUSR, Argument 5 VNUMBIN, Argument 6 VNUMSTRT, Argument 7 VNUMEND, Argument 8 RDEV) This plist maps to the X1200 DSECT that was shown in the HCPPLXIT COPY file on line 6589.</pre>	
000072 5900 C048 00048 12337 000072 5900 C048	<pre>C R0,=A(XIT@1200) Exit 1200?</pre>	Line 12337: A little defensive programming here. You could code the same entry point for more than one exit. Whether you do or not, it would be good practice to verify that R0 contains the exit number that you expect.

Table 12. Annotated Listing for Sample CP Exit Routine 1 for CP Exit 1200 (continued)

Assembler Source				Commentary
000076	A774	0044	000FE 12338	Line 12338 : Branch Relative instructions are used extensively in CP. Their use means a branch target may be well beyond the base register offset maximum of 4095, which then means modules may grow well beyond 4 KB yet require only a single base register for data. CP's mnemonic usage tends to be uppercase-B-lowercase-r-uppercase-condition, as in BrNE, BrC, BrU.
			BrNE DLEXIT ..no, leave	
			12340 *-----	
			12341 * If the original command was DIAL, or one of its	
			12342 * abbreviations, then we need to search the table for	
			12343 * the target userid.	
			12344 *	
			12345 * If the original command was not DIAL, then we need to	
			12346 * search the table for the command.	
			12347 *-----	
			179000001	
			181000001	
			182000001	
			183000001	
			184000001	
			185000001	
			186000001	
			187000001	
			188000001	

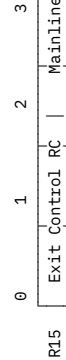
Table 12. Annotated Listing for Sample CP Exit Routine 1 for CP Exit 1200 (continued)

Assembler Source		Commentary	
00007A 5890 D01C	0001C 12349	L R9,SAVER1	Address of our plist
	12350	HCPUSING X1200,R9	Addressability
00007E 5810 900C	0000C 12354	L R1,X1200CMD	Address of command
000082 D504 C052 1000 00052	00000 12356	CLC =CL5'DIAL ',0(R1)	DIAL
000088 A784 C013	0000A 12357	BrE DL300	
00008C D503 C04C 1000 0004C	00000 12358	CLC =CL4'DIA ',0(R1)	DIA
000092 A784 C00E	0000A 12359	BrE DL300	
000096 D502 C057 1000 00057	00000 12360	CLC =CL3'DI ',0(R1)	DI
00009C A784 C009	0000A 12361	BrE DL300	
0000A0 D501 C050 1000 00050	00000 12362	CLC =CL2'D ',0(R1)	D
0000A6 A784 C004	0000A 12363	BrE DL300	
0000AA A7F4 C00E	0000C 12365	BrU DL400	Search table for cmd
	12367 *	-----*	
	12368 *	The command was DIAL. Therefore, search the table	*
	12369 *	* for the target userid.	*
	12370 *	-----*	
0000AE	12372 DL300	DS OH	Address of target userid
0000AE 5810 9010	00010 12373	L R1,X1200UID	Look for the target uid
0000BE A774 C020	000FE 13686	HCPLCALL SRCHUID	..not found
0000C2 A7F4 C00A	000D6 13687	BrNZ DLEXIT	..found
		BrU DL500	
	13689 *	-----*	
	13690 *	The command was not DIAL. Therefore, search the table	*
	13691 *	* for the command.	*
	13692 *	-----*	
0000C6	13694 DL400	DS OH	Address of the command
	13695 *cmt	L R1,X1200CMD	Look for the command
0000D2 A774 C016	13696	HCPLCALL SRCHCMD	..not found
	000FE 15008	BrNZ DLEXIT	..found
	15009 *cmt	BrU DL500	
	15011 *	-----*	
	15012 *	A table entry was found.	*
	15013 *	* Copy our control data to the exit 1200 plist.	*
	15014 *	-----*	
0000D6	15016 DL500	DS OH	Addressability
	15017	HCPUSING JAMTABLE,R7	

Line 12349: Here is an example of how to get to one of the arguments (in this case, the command that got us here) that was supplied to exit 1200. Remember that everything passed to an exit is passed by address.

Table 12. Annotated Listing for Sample CP Exit Routine 1 for CP Exit 1200 (continued)

Assembler Source		Commentary	
0000D6 5810 9010 00010 15021	R1,X1200UID	Return data back to the	Line 15021: Return data back to the mainline by using the addresses passed to you in the plist. Check the plist definition to be sure which entries you are allowed to change.
0000DA D207 1000 700C 00000 0000C 15022	0(L'VMDUSER,R1),JAMTUID	target uid	
0000E0 5810 9014 00014 15024	R1,X1200VDN	Return target vdev	Line 15024: Return target vdev
0000E4 D203 1000 7014 00000 00014 15025	0(4,R1),JAMTVDN	target vdev	
0000EA 5810 9018 00018 15027	R1,X1200VDS	Return vdev range start	Line 15027: Return vdev range start
0000EE D203 1000 7018 00000 00018 15028	0(4,R1),JAMTVDS	range start	
0000F4 5810 901C 0001C 15030	R1,X1200VDE	Return vdev range end	Line 15030: Return vdev range end
0000F8 D203 1000 701C 00000 0001C 15031	0(4,R1),JAMTVDE	range end	
15033 *cmt	DLEXIT	Return	Line 15033: Always return R15 = 0, unless you know for sure that it should not be. The value returned in R15 is really considered to be two unsigned halfword values: the exit control return code and the mainline return code.
15035 *-----*			
15036 * Return			
15037 *-----*			
0000FE 15039 DLEXIT DS 0H			Line 15039: Always return R15 = 0, unless you know for sure that it should not be. The value returned in R15 is really considered to be two unsigned halfword values: the exit control return code and the mainline return code.
0000FE 15039 DLEXIT DS 0H			
0000FE D203 D054 0A00 00054 00A00 15041	SAVER15,PEX0	Rc <- 0	Line 15041: Always return R15 = 0, unless you know for sure that it should not be. The value returned in R15 is really considered to be two unsigned halfword values: the exit control return code and the mainline return code.
0000FE D203 D054 0A00 00054 00A00 15042	HCPEXIT EP=(JANSAMD),SETCC=NO		
15600	HCPDROP R7		Line 15600: Always return R15 = 0, unless you know for sure that it should not be. The value returned in R15 is really considered to be two unsigned halfword values: the exit control return code and the mainline return code.
15602	HCPDROP R9		



Bytes 0-1 (the exit control return code) contain the directive to the exit controller. Bytes 0-1 will be set to zero by the exit control routine before being passed back to the mainline routine.

Bytes 2-3 (the mainline return code) contain the directive to the mainline routine. The exit controller will return to the mainline, as the return code in R15, the maximum mainline return code value from all of the exit routines that were called. The meaning of the return code will depend on the mainline routine.

Table 12. Annotated Listing for Sample CP Exit Routine 1 for CP Exit 1200 (continued)

Assembler Source	Commentary
<pre> 15605 * Start of specifications ***** 15606 * Entry Point Name - SRCHCMD 15607 * 15608 * Descriptive name - 15609 * Search the JAMTABLE entries for the first entry that 15610 * matches the argument passed in R1. 15611 * 15612 * Function - 15613 * Table lookup 15614 * 15615 * Register usage - 15616 * R0 - BCI loop reg for searching JAMTABLE 15617 * R1 - Address of 12 byte command 15618 * R2 - 15619 * R3 - 15620 * R4 - 15621 * R5 - 15622 * R6 - 15623 * R7 - Address of JAMTABLE 15624 * R8 - 15625 * R9 - 15626 * R10 - 15627 * R11 - Dispatched VMDBK address 15628 * R12 - Base register 15629 * R13 - Save Area address 15630 * R14 - Work register, linkage 15631 * R15 - Work register, linkage 15632 * 15633 * SAVEWRK usage - 15634 * SAVEWRK0 - 15635 * SAVEWRK1 - 15636 * SAVEWRK2 - 15637 * SAVEWRK3 - 15638 * SAVEWRK4 - 15639 * SAVEWRK5 - 15640 * SAVEWRK6 - 15641 * SAVEWRK7 - 15642 * SAVEWRK8 - 15643 * SAVEWRK9 - 15644 * 15645 * Input - 15646 * R1 - Address of 12 byte command 15647 * 15648 * Output - See Exit normal, Exit error 15649 * 15650 * </pre>	<pre> 26200001 26300001 26400001 26500001 26600001 26700001 26800001 26900001 27000001 27100001 27200001 27300001 27400001 27500001 27600001 27700001 27800001 27900001 28000001 28100001 28200001 28300001 28400001 28500001 28600001 28700001 28800001 28900001 29000001 29100001 29200001 29300001 29400001 29500001 29600001 29700001 29800001 29900001 30000001 30100001 30200001 30300001 30400001 30500001 30600001 30700001 </pre>
<pre> 15651 * Exit normal - 15652 * CC = 0, Table entry found 15653 * R7 = Address of JAMTABLE 15654 * 15655 * CC = 3, Table entry not found 15656 * 15657 * Exit error - 15658 * None 15659 * 15660 * General comments - 15661 * None 15662 * 15663 * End of specifications ***** </pre>	<pre> 30800001 30900001 31000001 31100001 31200001 31300001 31400001 31500001 31600001 31700001 31800001 31900001 32000001 </pre>

Table 12. Annotated Listing for Sample CP Exit Routine 1 for CP Exit 1200 (continued)

Assembler Source			Commentary		
15665 SRCHCMD HCPLENTR , 16136 *----- 16137 * Initialize: 16138 * SAVEWRK 16139 *-----			322000001 324000001 325000001 326000001 327000001	Line 15665 : HCPLENTR default entry type is CALL.	
16141 *cmt XC SAVEWRK,SAVEWRK 16143 *----- 16144 * Start looking for a table entry that matches the command 16145 * that R1 points to. 16146 *----- 00004 00004 LA R0,JAM\$DCN 00190 00190 LA R7,JAM\$DC 16150 HCPUSING JAMTABLE,R7 16154 SCMD100 DS 0H 00000 00000 CLC JAMTAMD,0(R1) 00140 00156 BIE SCMDCC0 00020 00158 LA R7,JAMTABLN(R7) 00126 00159 BICT R0,SCMD100 16161 SCMDCC3 DS 0H 16162 CC 3 00146 00164 BRU SCMDXIT 16166 SCMDCC0 DS 0H 00034 00167 ST R7,SAVER7 16168 CC 0 16170 00169 BRU SCMDXIT			329000001 331000001 332000001 333000001 334000001 336000001 337000001 338000001 340000001 341000001 342000001 344000001 345000001 347000001 348000001 349000001 351000001 352000001 353000001 354000001	HCPSVC actually does this Number of entries Address of first JAMTABLE Addressability Found it? ..yes Address of the next JAMTABLE Keep looking Set condition code = 3 Return Return address of JAMTABLE Set condition code = 0 Return	Line 16141 : A new CALL-type entry, HCPENTER or HCPLENTR, will be handed a new savearea. So, even though the same labels are used in this subroutine, they are touching storage different than the caller's savearea.

Table 12. Annotated Listing for Sample CP Exit Routine 1 for CP Exit 1200 (continued)

Assembler Source	Commentary
000146	
16172 *-----*	35600001
16173 * Return	35700001
16174 *-----*	35800001
16176 SCMDXIT DS OH	36000001
16177 HCPLEXIT ,	36100001
16736 HCPDROP R7	36300001
16739 * Start of specifications *****	36500001
16740 * Entry Point Name - SRCHUID	36600001
16741 * Descriptive name -	36700001
16742 * Search the JAMTABLE entries for the first entry that	36800001
16743 * matches the argument passed in R1.	36900001
16744 * Function -	37000001
16745 * Table lookup	37100001
16746 * Register usage -	37200001
16747 * R0 - BCT loop reg for searching JAMTABLE	37300001
16748 * R1 - Address of 8 byte user ID	37400001
16749 * R2 -	37500001
16750 * R3 -	37600001
16751 * R4 -	37700001
16752 * R5 -	37800001
16753 * R6 -	37900001
16754 * R7 - Address of JAMTABLE	38000001
16755 * R8 -	38100001
16756 * R9 -	38200001
16757 * R10 - Dispatched VMDBK address	38300001
16758 * R11 - Base register	38400001
16759 * R12 - Save Area address	38500001
16760 * R13 - Work register, linkage	38600001
16761 * R14 - Work register, linkage	38700001
16762 * R15 - Work register, linkage	38800001
16763 * R16 -	38900001
16764 * R17 -	39000001
16765 * R18 -	39100001
16766 * R19 -	39200001
16767 * R20 -	39300001
16768 * R21 -	39400001
16769 * R22 -	39500001
16770 * R23 -	39600001
16771 * R24 -	39700001
16772 * R25 -	39800001
16773 * R26 -	39900001
16774 * R27 -	40000001
16775 * R28 -	40100001
16776 * R29 -	40200001
16777 * R30 -	40300001
16778 * R31 -	40400001
16779 * R32 -	40500001

Table 12. Annotated Listing for Sample CP Exit Routine 1 for CP Exit 1200 (continued)

Assembler Source		Commentary	
	16780 * Input -		
	16781 * R1 - Address of 8 byte user ID		
	16782 *		
	16783 * Output - See Exit normal, Exit error		
	16784 *		
	16785 * Exit normal -		
	16786 * CC = 0, Table entry found		
	16787 * R7 = Address of JAMTABLE		
	16788 *		
	16789 * CC = 3, Table entry not found		
	16790 *		
	16791 * Exit error -		
	16792 * None		
	16793 *		
	16794 * General comments -		
	16795 * None		
	16796 *		
	16797 * End of specifications *****		
	16799 SRCHUID HCPLENTNTR ,		
	17270 *		
	17271 * Initialize:		
	17272 * SAVEWRK		
	17273 *		
	17275 *cmt XC SAVEWRK,SAVEWRK HCPSPVC actually does this		
	17277 *		
	17278 * Start looking for a table entry that matches the userid		
	17279 * that R1 points to.		
	17280 *		
00015E 4100 0004	00004 17282 LA R0,JAMSDCN	Number of entries	
000162 4170 C190	00190 17283 LA R7,JAMSDC	Address of first JAMTABLE	
	17284 HCPUSING JAMTABLE,R7	Addressability	
000166	17288 SUID100 DS 0H	Found it?	
00016C A784 000A	00000 17289 CLC JAMTUID,0(R1)	..yes	
	00180 17290 BIE SUIDCC0		
000170 4177 0020	00020 17292 LA R7,JAMTABLN(R7)	Address of the next JAMTABLE	
000174 A706 FFF9	00166 17293 BRCT R0,SUID100	Keep looking	
000178	17295 SUIDCC3 DS 0H		
00017C A7F4 0005	17296 CC 3	Set condition code = 3	
	00186 17298 BIE SUIDEXIT	Return	
000180	17300 SUIDCC0 DS 0H		
000180 5070 D034	00034 17301 ST R7,SAVER7	Return address of JAMTABLE	
	17302 CC 0	Set condition code = 0	
	17304 *cmt BIE SUIDEXIT	Return	
	17306 *		
	17307 * Return		
	17308 *		
000186	17310 SUIDEXIT DS 0H		
	17311 HCPLEXIT ,		
	17870 HCPDROP R7		

Table 12. Annotated Listing for Sample CP Exit Routine 1 for CP Exit 1200 (continued)

Assembler Source		Commentary	
17873	*-----*	Line 17873: These are our control	data definitions, which are mapped by JAMTABLE.
17874	* The JAMTABLE entries.		
17875	*-----*		
000190	17877 JAM\$DC DS 0D		
000190	17878 * CL12'?'	Notice the two entries for commands	
00019C	17879 DC CL8'HELPMACH'	BOOKS and JOURNALS, where the target	
0001A4	17880 DC F'-1'	user ID is the same but the range of target	
0001A8	17881 DC XL4'00000000'	virtual devices is different. This separation	
0001AC	17882 DC XL4'0000FFFF'	guarantees that neither the BOOKS users	
0001B0	17883 * CL12'BOOKS'	nor the JOURNALS users can consume all	
0001B0	17884 DC CL8'LIBRARY'	of the virtual terminal addresses, with one	
0001B0	17885 DC F'-1'	type of use locking out the other.	
0001B0	17886 DC XL4'00000000'	For this table to be most useful, you should	
0001C8	17887 DC XL4'00000000FF'	also enter the following DEFINE ALIAS	
0001C8	17888 * CL12'JOURNALS'	commands:	
0001C8	17889 DC CL8'LIBRARY'		
0001D0	17890 DC F'-1'		
0001D0	17891 DC XL4'000000100'		
0001D0	17892 DC XL4'0000001FF'		
0001E8	17893 * CL12'PVM'		
0001E8	17894 DC CL8'PVM'		
0001E8	17895 DC F'-1'		
0001F0	17896 DC XL4'00000000'		
0001F0	17897 DC XL4'0000FFFF'		
00020C	17898 * JAM\$DCN EQU (*JAM\$DC)/JAMTABLN		
00020C	17899 HCPDROP R0		
00020C	17900 HCPDROP R11		
00020C	17901 HCPDROP R13		
00020C	17902 HCPPEILG		
00020C	17903		
00020C	17904		
00020C	17905		
00020C	17906		
00020C	17907		
00020C	17908		
00020C	17909		
00020C	17910		
00020C	17911		

DEFINE ALIAS ? FOR DIAL ENABLE
DEFINE ALIAS BOOKS FOR DIAL ENABLE
DEFINE ALIAS JOURNALS FOR DIAL ENABLE
DEFINE ALIAS PVM FOR DIAL ENABLE

If so, then any of the aliases (? , BOOKS, JOURNALS, PVM) or the base command (DIAL) will cause HCPDIA to be called, and HCPDIA will call exit 1200.

If a user at a terminal enters the command '?', or 'BOOKS', or any of the other alias names, it will be treated as an alias to DIAL, which means that CP will treat it just as if the user had entered the command 'DIAL' with no operands. Unless exit 1200 supplies a target user ID, this command will fail. It is up to exit 1200 to make the command 'whole'.

Table 13. Annotated Listing for Sample CP Exit Routine 2 for CP Exit 1200

Assembler Source		Commentary
12	MACRO	Line 12: This macro, JAMMDLAT, describes the modules and entry points for component ID JAM. This macro can be included inline, as shown here, or in a MACLIB that is processed by the compiler. The HPCMPID macro, invoked on line 19, adds JAM to the component IDs for this compile.
13	JAMMDLAT &EPNAME	
14	MDLATHDR &EPNAME	
15	MDLATENT JAMSAM,MODATTR=(MP,DYN),	
	CPXLOAD=YES	
16	MDLATTLR	
17	MEND	
		00600001 00650001 00700001 X00750001 00800001 00850001 00900001

Table 13. Annotated Listing for Sample CP Exit Routine 2 for CP Exit 1200 (continued)

Assembler Source		Commentary
19 20	HPCMPID COMPID=JAM COPY HCPOPTMS	<p>Line 19: HPCMPID is a macroinstruction whose purpose is to specify your component ID entries. CP uses the component ID entries to identify your xxxMDLAT macroinstructions, which CP uses to assign linkage attributes. Component ID entries are 3-character strings. For example, IBM's component ID for CP is 'HCP'.</p> <p>Component ID entries can also be used on HPCONSL MACROs, which would connect the HPCONSL MACRO with associated message repositories (see the CP ASSOCIATE MESSAGE command for more information). HPCMPID may be specified multiple times, and with multiple component ID entries. For example:</p> <div>HPCMPID COMPID=(JAM,JAM,XYZ) HPCMPID COMPID=IJK</div> <p>The HPCMPID MACRO is tied to the support for multiple HCPMDLAT MACROs, each known as xxxMDLAT. The 'xxx' letters represent the component ID entries specified in your HPCMPID MACROs. Based on the HPCMPID MACROs shown above, the xxxMDLAT MACROs that HPCALL would use are:</p> <div>HCPMDLAT MACRO always checked first JAMMDLAT MACRO JAMMDLAT MACRO XYZMDLAT MACRO IJKMDLAT MACRO</div> <p>JAMMDLAT is shown twice because JAM is specified twice in the HPCMPID MACROs.</p>

Table 13. Annotated Listing for Sample CP Exit Routine 2 for CP Exit 1200 (continued)

Assembler Source	Commentary
	Line 19 (continued): HPCALL and other MACROs call your xxxMDLAT MACRO looking for the attributes of the module in order to know the proper linkage to generate. HPCALL will call your xxxMDLAT MACROs in the order that your component ID entries are specified on the HCPCMPID MACROs.
	The format of your xxxMDLAT MACROs is similar to the format of the HCPMDLAT MACRO. Here is our JAMMDLAT MACRO, also shown in the listing at line 12:
	<pre>MACRO JAMMDLAT, SERNAME JAMMDLAT SERNAME MDLATENT JAMMDLAT MODATTR=(MP,DYN), X MDLATTLR CPXLOAD=YES MEND</pre>
	You use the MDLATENT MACRO to specify the attributes of your module using the same keywords as you used in HCPMDLAT. For a typical module, what you see here is all that you would need. Only if your module has special attributes beyond these would you investigate other MDLATENT keywords.
	Attribute: Meaning:
	MP - Multiprocessor capable. This means that tasks on different processors may run this module simultaneously.
	DYN - Uses a dynamic savearea (SAVBK).
	Attributes, specifically MP or NONMP , specified here must match the attributes specified by CPXLOAD.

Table 13. Annotated Listing for Sample CP Exit Routine 2 for CP Exit 1200 (continued)

Assembler Source		Commentary	
000000 000000 000000 000002 000004 000008 000000	347 JAMSAM	HCPPROLG ATTR=(RESIDENT,REENTERABLE), COPYR=(1995,2005),COPYRID='My Copyright', BASE=(R12)	X01150001 X01200001 01300001
	000000 000000 00AE8		
	480+JAMSAM	CSECT ,	&VXIN0ZW 01 - HCPPR
	481+HCP@MOD DS	0D	@VXIN0ZW 01 - HCPPR
	482+ DC AL2(C'V')	Bogus instruction to prevent fall-thru	@VRGB1QY 01 - HCPPR
000000 00E5 000002 E5E5 000004 00000000 000008 00000000	483+ DC C'VV'	For HCPENTER enforcement characters	@VRGB1QY 01 - HCPPR
	484+ DC A(0)	No call - or goto-by-register vector	@VRGB1QY 01 - HCPPR
	485+ DC A(0)	For PLX, branch around prologue	@VRGB1QYX01 - HCPPR
	+	For ASM, spacer	@VRGB1QY
00000C 918194A28194A040 000016 C5D4C5F2F0F50000 00001E F0F761F2F961F0F5 000026 7AF1F04BF4F5D9C5	486+	DC CL8'jamsam',AL2((HCP@END-HCP@MOD)/8)	@VR4GCZW 01 - HCPPR
	487+	DC CL6'EME205',XL2'0000'	@A0002FA 01 - HCPPR
	488+	DC CL8'07/29/05'	@A0002FA 01 - HCPPR
	489+	DC CL6':10.45',CL1'R',CL1'E'	@A0002FA 01 - HCPPR
00002E D4A840C39697A899 00003A F1F9F9F5 00003E 6B 00003F F2F0F0F4 000048	490+	DC C'My Copyright'	@VRSH1QY 01 - HCPPR
	491+	DC CL4'1995'	@VR8G0U2 01 - HCPPR
	492+	DC CL1','	@VR8G0U2 01 - HCPPR
	493+	DC CL4'2005'	@VR8G0U2 01 - HCPPR
	494+HCP@0002	DS 0D LABEL FOR BRANCH AROUND PROLG	@P6839ZW 01 - HCPPR

Line **347**: The keyword 'COPYRID=' is used only if the keyword 'COPYR=' is specified. The string assigned to 'COPYRID=' will be assembled as a character constant in place of the IBM copyright notice.

Lines **482-485**: Here you see "bogus instruction", enforcement characters, address of vector, and so on. These fields mirror fields generated by HCPENTER to support call-by-register and goto-by-register. None of these data fields are executable instructions, so execution cannot fall through into any HCPENTER macro, which is why there is a "bogus instruction".

Line **486**: The module name is saved in lower case whenever it does not start with 'HCP'. Services in CP may use this value as stored here (TRACE instructions for HPCALL/HCPEXIT linkage) or may convert it to upper case (HCPCONSL when building error message headers). This is just another peculiarity that you should be aware of.

Lines **490-493**: You can use the 'COPYR=' *and* 'COPYRID=' keywords to insert your own copyright information into the expansion of the HCPPROLG macro. Notice that both keywords must be specified, or else your copyright information will not be generated.

Table 13. Annotated Listing for Sample CP Exit Routine 2 for CP Exit 1200 (continued)

Assembler Source		Commentary							
000048 000060	571+HCPDATAA LOCTR , 573+HCPCODEA LOCTR , 575 PRINT ON,NOGEN	@Y0656QY 02 - HCPDA @Y0656QY 02 - HCPDA 01350001	Lines 571-573 : LOCTR instructions are used to place data early in the module (in this example, data starts at offset 0x000048) followed by executable code (in this example, code starts at offset 0x000060). Certain macros will generate LOCTR instructions in order to add their data to these sections. So you may find instructions that are sequential in the listing but are not sequential in memory. There will be more examples of this shown below.						
	577 HCPXTRN HCPCVTRM - Bin to decimal conversion, trim 0's 579 HCPXTRN HCPCVUBM - Binary to Filemode conversion 581 HCPXTRN HCPZICLS - Close file on a CP-accessed mdisk 583 HCPXTRN HCPZIGET - Get a record from a CMS file 585 HCPXTRN HCPZIOPN - Open file on CP-accessed minidisk 587 HCPXTRN HCPZPRPG - General parser	01450001 01500001 01550001 01600001 01650001 01700001							
	590 COPY HCPBITMP - Bit Map Control Block 710 COPY HCPCMPBK - Component ID Block 829 HCPCONDF - Statement definition mapping 1002 HCPCSLPL - Console MACRO parameter list 1924 HCPDRBK - Data Request Block 2623 HCPQUAT - General equates	01800001 01850001 01900001 01950001 02000001 02050001							
	5501 HCPQXIT - Equates for Exit control 5662 COPY HCPGSDBK - General System Data Block 5847 COPY HCPMXRBK - Message number equates 10865 HCPPFXPG - Host Prefix Page	02100001 02150001 02200001 02250001							
Line 5501 : HCPQXIT is a good place to look for exit equates. Exit equates should all be of the form: XIT@xxx EQU X'xxx'. As always, we suggest that you do not add your exit number equates to this IBM COPY file. Add them to your own COPY file instead. IBM-defined CP exit routines should have equates of this form in the HCPQXIT COPY file. The intent is that every exit point should be listed here, so that we can tell what has been used, with a small amount of information about what it is for. The two exits in DIAL processing are assigned these CP exit numbers:									
		<table><tr><td>XIT@1200 EQU</td><td>X'1200'</td><td>Command: DIAL Validate operands passed on the DIAL command</td></tr><tr><td>XIT@1201 EQU</td><td>X'1201'</td><td>Command: DIAL Second chance, after target has been decided.</td></tr></table>		XIT@1200 EQU	X'1200'	Command: DIAL Validate operands passed on the DIAL command	XIT@1201 EQU	X'1201'	Command: DIAL Second chance, after target has been decided.
XIT@1200 EQU	X'1200'	Command: DIAL Validate operands passed on the DIAL command							
XIT@1201 EQU	X'1201'	Command: DIAL Second chance, after target has been decided.							

Table 13. Annotated Listing for Sample CP Exit Routine 2 for CP Exit 1200 (continued)

Assembler Source		Commentary	
0000000	13838	COPY HCPPLXIT - PLIST definitions for IBM Exit Points	023000001
	X1200	DSECT ,	70500210
	*		71000210
	*	The PLIST pointers for exit 1200	71500210
		Read all comments here as if	72000210
		they all begin "Address of"	72500210
0000000	X1200UMD DS	1 Reserved	73000210
0000004	X1200TRT DS	2 User words	73500210
0000008	X1200CMD DS	3 TRT 256 bytes	74000210
000000C	X1200UTD DS	4 Command name	74500210
0000010	X1200VDN DS	5 Target userid	75000210
0000014	X1200VDS DS	6 Target VDEV number	75500210
0000018	X1200VDE DS	7 VDEV number start search range	76000210
000001C	X1200RVD DS	8 VDEV number end search range	
0000020		9 RDEV	
14224	COPY HCPRDEV - Real Device Block		023500001
15115	COPY HCPSAVBK - Savearea Block		024000001
15482	COPY HCPSSBDF - Syntax Subdefinition description area		024500001
15615	COPY HCPSSPBK - Pointer to SSBDF		025000001
15742	COPY HCPSTADF - Syntax state definition area		025500001
15863	COPY HCPSYNDF - Syntax definition area		026000001
16040	COPY HCPSYSCM - Host System Common Area		026500001
17439	COPY HCPTOKDF - Token definition mapping		027000001
17850	COPY HCPVMDBK - Virtual Machine Definition Block		027500001
22112	COPY HCPXCIBK - CP Exit Control Block		028000001
22290	HCPUSING PEXPG,R0		029000001
22293	HCPUSING VMDBK,R11		029500001
22296	HCPUSING SAVBK,R13		030000001

Line **13838**: HCPPLXIT is the place to look for plist definitions for IBM-defined CP exit points. As always, we suggest that you do not add your exit plist definitions to this IBM COPY file. Add them to your own COPY file instead. Every exit plist should be of the form:

```

Xxxxx DSECT A
DS A
XxxxxUMD DS A
... words doubleword aligned
... initialized to binary

XxxxxTRT DS A
... zeros
... Address of 256 bytes
... doubleword aligned
... initialized to binary

Xxxxx... DS A
Xxxxx... DS A
... zeros
Xxxxx... DS A
Address of a data item
Xxxxx... DS A
Address of a data item

```

Storage for the plist is allocated in the mainline routine for each instance of calling an exit. The plist and what it points to are untouched by the exit control routines. Each exit sees whatever the prior exit left. The storage is deleted upon final return from all of the exit routines.

Plist addresses after the first three standard entries all depend on the exit. In fact, the exit need not conform to using the first three standard entries. How the mainline builds the plist for its exit may be rather arbitrary.

Table 13. Annotated Listing for Sample CP Exit Routine 2 for CP Exit 1200 (continued)

Assembler Source			Commentary	
00000000 00000000	000000 000024 000000	22300 JAMTABLE DSECT , 22301 JAMTFWD DS A 22302 JAMTGSD EQU JAMTFWD	My command-to-userid table Address of next JAMTABLE Address of error msg GSOBK ... if the parser detected ... illegal syntax	031000001 031500001 X032000001 X032500001 033000001 033500001 034000001 034500001 035000001 035500001 036000001
00000004 0000010 0000018 000001C 0000020	000024	22303 JAMTCMD DS CL12 22304 JAMTUID DS CL(L'VMDUSER) 22305 JAMTVDN DS F 22306 JAMTVDS DS F 22307 JAMTVDE DS F 22308 JAMTABLN EQU *-JAMTABLE	Command Target userid Target VDEV number VDEV number range start VDEV number range end	
Line 22300: This is our local DSECT. It maps the storage that the parser fills in (at line 49151) based on the parser syntax definition at line 70492.				
<div>22310 * Start of specifications ***** 22311 * Entry Point Name - JAMSAMD 22312 * 22313 * Descriptive name - 22314 * Dial exit 1200 22315 * 22316 * Function - 22317 * Preprocess DIAL command input 22318 * 22319 * Register usage - 22320 * R0 - Exit number 22321 * R1 - Address of exit plist pointers 22322 * R2 - 22323 * R3 - 22324 * R4 - Address of argument 4, the command 22325 * R5 - Address of argument 5, the target userid 22326 * R6 - 22327 * R7 - Address of JAMTABLE 22328 * R8 - 22329 * R9 - Address of exit 1200 plist 22330 * R10 - Address of our CMPBK 22331 * R11 - Dispatched VMOBK address 22332 * R12 - Base register 22333 * R13 - Save Area address 22334 * R14 - Work register, linkage 22335 * R15 - Work register, linkage 22336 * 22337 * SAVEWRK usage - 22338 * SAVEWRK0 - Flags 22339 * SAVEWRK1 - 22340 * SAVEWRK2 - 22341 * SAVEWRK3 - 22342 * SAVEWRK4 - 22343 * SAVEWRK5 - 22344 * SAVEWRK6 - 22345 * SAVEWRK7 - 22346 * SAVEWRK8 - 22347 * SAVEWRK9 - 22348 * 22349 *</div> <div>037000001 037500001 038000001 038500001 039000001 039500001 040000001 040500001 041000001 041500001 042000001 042500001 043000001 043500001 044000001 044500001 045000001 045500001 046000001 046500001 047000001 047500001 048000001 048500001 049000001 049500001 050000001 050500001 051000001 051500001 052000001 052500001 053000001 053500001 054000001 054500001 055000001 055500001 056000001 056500001</div>				

Table 13. Annotated Listing for Sample CP Exit Routine 2 for CP Exit 1200 (continued)

Assembler Source	Commentary
<pre> 22350 * Input - 22351 * R0 - Exit number 22352 * R1 - Address of plist addresses 22353 * R2 - Address of XCRBK 22354 * 22355 * Output - See Exit normal, Exit error 22356 * 22357 * Exit normal - 22358 * R15 = 0 22359 * Target userid, VDEV changed 22360 * 22361 * Exit error - 22362 * None 22363 *</pre>	<p>Line 22350: The standard contents of the registers at entry to all exit routines will be like this:</p> <ul style="list-style-type: none"> • R0 will contain the exit number. • R1 will contain the address of an array of addresses (the plist). The high order bit of the last address will be on. • R2 will contain the address of an XCRBK (Exit Call Request Block). • R3-R10 will be garbage. • R11 may contain the address of the dispatched VMDBK, or maybe not. That all depends on the exit. • R12 will be used as the module base register. • R13 will be used as the SAVBK base register. • R14-R15 will be used as linkage registers. <p>Storage for the XCRBK is allocated or deleted by ASSOCIATE EXIT processing. An XCRBK is allocated for each epname specified by ASSOCIATE EXIT. If an epname is specified more than once, each occurrence gets a separate XCRBK. The user words (Reserved for non-IBM use) are initialized to binary zeros when allocated but are otherwise untouched by the exit control routines.</p>

Table 13. Annotated Listing for Sample CP Exit Routine 2 for CP Exit 1200 (continued)

Assembler Source		Commentary
		Line 22350 (continued): In this example, we do not use the HCPXCRBK, so we do not show it in the listing. Here is what it looks like:
XCRBK	DSECT ///// (various control fields)	
*		
XCRUSD1 DS	D	Reserved for non-IBM use
XCRUSD2 DS	D	Reserved for non-IBM use
XCRUSRF1 DS	F	Reserved for non-IBM use
XCRUSRF2 DS	F	Reserved for non-IBM use
XCRUSRH1 DS	H	Reserved for non-IBM use
XCRUSRH2 DS	H	Reserved for non-IBM use
XCRUSRX1 DS	X	Reserved for non-IBM use
XCRUSRX2 DS	X	Reserved for non-IBM use
XCRUSRX3 DS	X	Reserved for non-IBM use
XCRUSRX4 DS	X	Reserved for non-IBM use
*		
XCRFWD DS	A	Address of next XCRBK
*		
XCRATMPT DS	F	Count of attempts to call this routine
*		
XCRCALLS DS	F	Count of calls completed
*		
DS	F	Reserved
XCRMSACT DS	D	Time (in micro-seconds) that ... this routine was active.
*		
XCR\$END DS	00	The end
*		

Assembler Source

Commentary

Line 22365: This plist maps to the X1200 DSECT that was shown in the HCPPLXIT COPY file on line 13838.

22364 *	* General comments -	*	06400001
22365 *	* The original plist built for exit 1200 was this:	*	06450001
22366 *		*	06500001
22367 *	PLIST=('RESERVED', Argument 1	*	06550001
22368 *	'USERWORD', Argument 2	*	06600001
22369 *	'TRITTABLE', Argument 3	*	06650001
22370 *	SAYCMD, Argument 4	*	06700001
22371 *	DIALEUSR, Argument 5	*	06750001
22372 *	VNUMBIN, Argument 6	*	06800001
22373 *	VNUMSTRT, Argument 7	*	06850001
22374 *	VNUMEND, Argument 8	*	06900001
22375 *	RDVE),	*	06950001
22376 *		*	07000001
22377 *		*	07050001
22378 *		*	07100001
22379 *		*	07150001
22380 *		*	07200001
22381 *	Our purpose is to customize the DIAL command	*	07250001
22382 *	processing. Given some control data (either the	*	07300001
22383 *	command name that caused the DIAL processing to be	*	07350001
22384 *	driven or the target userid), we will customize the DIAL	*	07400001
22385 *	processing. To do so, we need a table that associates	*	07450001
22386 *	the incoming command name with the data to return to	*	07500001
22387 *	DIAL. We will keep the source of that table on disk.	*	07550001
22388 *	We will load that table based on user instructions. We	*	07600001
22389 *	will read the table from disk and place it in storage.	*	07650001
22390 *	We will search the in-storage data as needed.	*	07700001
22391 *		*	07750001
22392 *	One interesting aspect of this sample is that we have	*	07800001
22393 *	defined an exit (number F200) for exit 1200. By doing	*	07850001
22394 *	this, we give the user the ability to tell us to reload	*	07900001
22395 *	the table. When the user enables exit F200, then the	*	07950001
22396 *	next time exit 1200 is called, it will notice that exit	*	08000001
22397 *	F200 is enabled and will call it. Exit F200 will	*	08050001
22398 *	reload the table. Before reloading the table, exit	*	08100001
22399 *	F200 will disable itself, so that the table will be	*	08150001
22400 *	loaded only once for each time exit F200 is enabled.	*	08200001
22401 *	And if the user enables exit 1200 (to cause us to be	*	08250001
22402 *	called) and exit F200 (to cause us to load the table)	*	08300001
22403 *	by using statements in the SYSTEM CONFIG file, we will	*	08350001
	have the table automatically loaded the first time DIAL	*	
	is invoked.	*	

Assembler Source

Line **23046**: A little defensive programming here. You could code the same entry point for more than one exit. Whether you do or not, it would be good practice to verify that R0 contains the exit number that you expect.

Table 13. Annotated Listing for Sample CP Exit Routine 2 for CP Exit 1200 (continued)

Assembler Source				Commentary
00017E A774 00AA	002D2 23047	BrNE DLEXIT	..no, leave	Line 23047: Branch Relative instructions are used extensively in CP. Their use means a branch target may be well beyond the base register offset maximum of 4095, which then means modules may grow well beyond 4 KB yet require only a single base register for data. CP's mnemonic usage tends to be uppercase-B-lowercase-r-uppercase-condition, as in BrNE, BrC, BrU.
	23049 *	-----	-----*	
	23050 *	Test exit number F200. If enabled, call it.	-----*	
	23051 *	Even if the call fails, our component ID block may have	-----*	
	23052 *	some useful data, so continue normally.	-----*	
23055	23053 *	-----	-----*	Line 23055: An exit number can be specified explicitly, as is done here. Or an exit number can be supplied in a register, for example 'EXIT=(R0)'. X11750001 11800001
		HCPXSERV TEST,EXIT=F200,		
		DISABLED=DL100		

Table 13. Annotated Listing for Sample CP Exit Routine 2 for CP Exit 1200 (continued)

Assembler Source		Commentary	
0001A6 5810 D01C	0001C 23068	L R1,SAVER1	Line 23068: Here we call an exit passing the plist that was already built for exit 1200. Since we are now considered the mainline for exit F200, we can pass R1 with anything that we want, and we can use the return code from exit F200 any way that we want.
	23069	HCPXSERV CALL,EXIT=F200,	
		PLISTBLD=(R1),	
		ERROR=DL100	
0001C0	24660 DL100 DS 0H		
	24662 *	*****	
	24663 *	Get our component ID block. It should have been filled	
	24664 *	in by exit F200, but we won't assume so.	
	24665 *		
	24666 *	If our component ID block does not exist (which is	
	24667 *	indicated by R15 = 4 from FIND), then we have nothing to	
	24668 *	do.	
	24669 *		
	24670 *	If our component ID block does exist (which is indicated	
	24671 *	by R15 = 0 from FIND), then we have something to do. We	
	24672 *	will get shared control over the component ID block, so	
	24673 *	that no changes may be made until we finish. We will	
	24674 *	perform our table search, and so forth. Then, we will	
	24675 *	relinquish our control over the component ID block and	
	24676 *	return.	
	24677 *	*****	

```
HCPXSERV CALL,EXIT=1200,
PLIST=( 'RESERVED',
'USERWORD',
'TRTTABLE',
SAVCMO,
DIALEUSR,
VNUMBIN,
VNUMSTRT,
VNUMEND,
RDEV),
PLISTBLD='GETST',
ERROR=EXX01Z
R1 -> the plist
Argument 1 i/w
Argument 2 i/w
Argument 3 i/o
Argument 4 i/w
Argument 5 i/w
Argument 6 i/w
Argument 7 i/w
Argument 8 i/w
Argument 9 i/o
R1 -> the plist
```

'PLISTBLD=(R1)' that we use means that the parameter list has already been built, and that R1 points to it. HCPDIA built it, and we can pass it on.

'PLISTBLD='GETST'" that HCPDIA used meant that a new parameter list was to be built in acquired storage, and the address of that storage was to be returned in R1.

Table 13. Annotated Listing for Sample CP Exit Routine 2 for CP Exit 1200 (continued)

Assembler Source			Commentary
24679	HCPXSERV FTND, COMPID= 'JAM'	13100001	<p>Line 24679: This is an example for finding your component ID block (CMPBK). A CMPBK is:</p> <ul style="list-style-type: none">• A control block (32 bytes of user data fields by default) that you can use as system-wide fields for your additions to CP• Identified by a unique 3-character component ID• Allocated for you by HCPXSERV MACRO• Deallocated for you by HCPXSERV MACRO• Serialized for you by HCPXSERV MACRO• A place to use to avoid changes to SYSCM <p>Your component ID block can be your anchor for any control blocks that your exits or mods need. You control access to the CMPBK by using the HCPXSERV MACRO. The CMPBK is an attempt to provide a mechanism for you to have system-wide anchor for your additions to CP without requiring local modifications to SYSCM.</p> <p>Here are the HCPXSERV MACROS related to component ID entries:</p>
<div>HCPXSERV ALLOCATE, COMPID=..... HCPXSERV DEALLOCATE, COMPID=..... HCPXSERV FTND, COMPID=..... HCPXSERV LOCKEXCL, COMPID=..... HCPXSERV LOCKEXCLUSIVE, COMPID=..... HCPXSERV LOCKSHARED, COMPID=..... HCPXSERV LOCKSHR, COMPID=..... HCPXSERV UNLOCKEXCL, COMPID=..... HCPXSERV UNLOCKEXCLUSIVE, COMPID=..... HCPXSERV UNLOCKSHARED, COMPID=..... HCPXSERV UNLOCKSHR, COMPID=.....</div>			

Table 13. Annotated Listing for Sample CP Exit Routine 2 for CP Exit 1200 (continued)

Assembler Source

Commentary

Line 24679 (continued): The CMPBK DSECT looks like this:

CMPBK	DSECT	Component ID Block
*		
*		-----*
*		* The IBM fields should remain at the front *
*		-----*
CMPBKLK	DS 30	Lock word to serialize access to this CMPBK
*		-----*
CMPFWD	DS A	Address of the next CMPBK
*		
CMPEXTNQ	DS FL1	Size of extension at CMPEXTND in Number of Quad-words. The maximum allowed value for CMPEXTNQ is 240, which gives a maximum size for a CMPBK of x'60' + 240*16 = 3936 bytes which allows for a little growth up to 4072 bytes.
*		
CMPID	DS CL3	The component Id
*		
	DS CL8	Reserved for IBM use
	DS CL8	Reserved for IBM use
	DS CL8	Reserved for IBM use
	DS CL8	Reserved for IBM use
*		-----*
*		* The user fields should remain at the end *
*		-----*
CMPUSRD1	DS D	Reserved for non-IBM use
CMPUSRD2	DS D	Reserved for non-IBM use
CMPUSRF1	DS F	Reserved for non-IBM use
CMPUSRF2	DS F	Reserved for non-IBM use
CMPUSRH1	DS H	Reserved for non-IBM use
CMPUSRH2	DS H	Reserved for non-IBM use
CMPUSRX1	DS X	Reserved for non-IBM use
CMPUSRX2	DS X	Reserved for non-IBM use
CMPUSRX3	DS X	Reserved for non-IBM use
CMPUSRX4	DS X	Reserved for non-IBM use
*		
	SPACE 5	
CMPEXTND	DS 00	Start of variable length data
*		The CMPBK may be extended at allocation time in units of quad-words (16 bytes)
*		
*		

The user fields (Reserved for non-IBM use) are there for you to anchor pointers to your other control blocks. Because the ownership of additions (yours or vendors) to CP should be identified with a unique component ID, there should not be collisions in the use of these control blocks.

Table 13. Annotated Listing for Sample CP Exit Routine 2 for CP Exit 1200 (continued)

Assembler Source				Commentary	
0001CE 12FF	001D9 A784 000E	26261	HPCASRC	(DL200, DLEXIT), R15=RJ15	Line 26261 : Here is an example of a macro expansion that generates instructions out-of-line. Line 26276 shows a LOCTR statement that causes the next instruction to be placed at offset 0x000118. Following the branch table, line 26282 shows a LOCTR statement that causes the next instruction, namely the definition of label 'DL200', to be placed at offset 0x0001EC.
0001D9 A784 000E	BzZ	26266+	LTR	00: found	
0001D4 A744 007F	BH	001EC 26267+	BzZ	04: not found	
0001D8 A7FE 0008	BH	002D2 26268+	BH	08: bad argument passed	
0001DC A724 007B	CHI	00008 26269+	CHI	?: invalid rc	
0001E0 A7F1 0003	BH	002D2 26270+	BH	Check for zero or negative.	
0001E4 A774 0077	TWL	00003 26271+	TWL	Fast path for RC0	
0001E8 A77F C118	BzZ	002D2 26272+	BzZ	Minus, unexpected ret code	
000118 00AE8 26276+HPCODE5 LOCTR	B	00118 00AE8 26273+	B	Within branch table?	
000118 26277+CASRC0580	0F	002D2 26274+	0F	No, unexpected return code	
000118 A7F4 006A	BzU	001EC 26278+	BzU	Return code multiple of 4?	Line 26285 : These are the two different ways (forgetting about alternative spellings) to get control over your component ID block:
00011C A7F4 00DB	BzU	002D2 26279+	BzU	No, bad return code	
000120 A7F4 00D9	BzU	002D2 26280+	BzU	Branch into branch table	
0001EC	LOCTR	00164 00AE8 26282+HPCODEA	LOCTR	Branch table, word aligned	
0001EC	DL200	26283	DL200	00	
				04	
				08	
				00	
				04	
				08	
000218 18A1	DS	26285	HCPXSERV	LOCKSHARED,COMPID='JAM'	It is a very good idea to lock access to any CMPBK before you attempt to use it because it might disappear (by HCPXSERV DEALLOCATE). When it comes time to relinquish the lock over your CMPBK, be sure to use the complementary UNLOCK request.
00021A 9602 D058	LR	27888	DL250	00: success	
	HCPUSING CMPBK,R10	27889	R10,R1	04: not found	
	OI SAVEWRK0,X'02'	27893	OI	08: passed bad R1	
				12: lock was deleted	
				16: something else bad	
				?: invalid rc	
				Address of our CMPBK	
				Addressability	
				CMPBK is locked shared	
		27895 *		*****	
		27896 *		If the original command was DIAL, or one of its	
		27897 *		abbreviations, then we need to search the table for	
		27898 *		the target userid.	
		27899 *			
		27900 *		If the original command was not DIAL, then we need to	
		27901 *		search the table for the command.	
		27902 *		*****	

Table 13. Annotated Listing for Sample CP Exit Routine 2 for CP Exit 1200 (continued)

Assembler Source		Commentary	
000282	D207 1000 7010 00000 00010 30587	MVC 0(L'VMDUSER,R1),JAMTUID	17400001
000288	5800 9020	L R0,X1200RDV	17500001
00028C	5800 7018	HCPUSING RDEV,R8	17500001
000290	957E 7018	L R0,JAMTVDS	17600001
000294	A774 0011	CLI JAMTVDS,C'='	17650001
000298	5800 0A48	BTRNE DL550	17700001
00029C	D503 8130 0A00 00130	L R0,PEXFFS	17750001
0002A2	A774 000A	CLC RDEV,SNA,PF0	17800001
0002A6	D503 8014 0A00 00014	CLC RDEVLSOP,PF0	17850001
0002AC	A774 0005	BTRNE DL550	17900001
0002B0	1F00	SLR R0,R0	17950001
0002B2	BF03 80A8	ICM R0,B'0011',RDEVDEV	18000001
0002B6	5810 9014	DS 0H	18050001
0002BA	5000 1000	L R1,X1200VDN	18100001
		ST R0,0(R1)	18150001
		HCPDROP R8	18200001
0002BE	5810 9018	L R1,X1200VDS	18250001
0002C2	D203 1000 701C 00000 0001C 30610	MVC 0(4,R1),JAMTVDS	18350001
0002C8	5810 901C	L R1,X1200VDE	18400001
0002CC	D203 1000 7020 00000 00020 30613	MVC 0(4,R1),JAMTVDE	18500001
		BRI DLEXIT	18550001
		*cmt	18600001
		30616 *	18700001
		30617 * Return	18750001
		30618 *	18800001
0002D2	A785 0267	DS 0H	18900001
0002D2	A785 0267	BIAS R8,LDUNLOCK	18950001
		Unlock our CMPBK	

Line **30587**: Return data back to the mainline by using the addresses passed to you in the plist. Check the plist definition to be sure which entries you are allowed to change.

Table 13. Annotated Listing for Sample CP Exit Routine 2 for CP Exit 1200 (continued)

Assembler Source				Commentary	
0002D6	D203	D054	0A00 00054 00A00	MVC SAVER15,PFx0	Line 30623: Always return R15 = 0, unless you know for sure that it should not be. The value returned in R15 is really considered to be 2 unsigned halfword values: the exit control return code and the mainline return code.
0002DC	58FD	0014	00014 00014	HCPEXIT EP=(JAMSAMD),SETCC=NO	
0002E0	00EF			L R15,SAVERETN-SAVBK(R13,0)	
				BASR R14,R15	
					R15 0 1 2 3 Exit Control RC Mainline RC
31182				HCPDROP R7	
31184				HCPDROP R9	
31186				HCPDROP R10	
31189	*			* Start of specifications *****	Bytes 0-1 (the exit control return code) contain the directive to the exit controller. Bytes 0-1 will be set to zero by the exit control routine before being passed back to the mainline routine.
31190	*			* Entry Point Name - SRCHCMD	
31191	*			* Descriptive name -	
31192	*			* Search the chain of JAMTABLE control blocks for the first entry whose JAMTCMD entry matches the argument passed in R1.	
31193	*			* Function -	Bytes 2-3 (the mainline return code) contain the directive to the mainline routine. The exit controller will return to the mainline, as the return code in R15, the maximum mainline return code value from all of the exit routines that were called. The meaning of the return code will depend on the mainline routine.
31194	*			* Table lookup	
31195	*			* Register usage -	
31196	*			* R0 - Address of 12 byte command	
31197	*			* R1 -	
31198	*			* R2 -	
31199	*			* R3 -	
31200	*			* R4 -	
31201	*			* R5 -	
31202	*			* R6 -	
31203	*			* R7 -	
31204	*			* R8 -	
31205	*			* R9 -	
31206	*			* R10 -	
31207	*			* R11 -	
31208	*			* R12 -	
31209	*			* R13 -	
31210	*			* R14 -	
31211	*			* R15 -	
31212	*			* R16 -	
31213	*			* R17 -	
31214	*			* R18 -	
31215	*			* R19 -	
31216	*			* R20 -	
31217	*			* R21 -	
31218	*			* R22 -	
				* R23 -	
				* R24 -	

Table 13. Annotated Listing for Sample CP Exit Routine 2 for CP Exit 1200 (continued)

Assembler Source	Commentary
<pre> 31219 * SAVEMRK usage - 31220 * SAVEMRK0 - 31221 * SAVEMRK1 - 31222 * SAVEMRK2 - 31223 * SAVEMRK3 - 31224 * SAVEMRK4 - 31225 * SAVEMRK5 - 31226 * SAVEMRK6 - 31227 * SAVEMRK7 - 31228 * SAVEMRK8 - 31229 * SAVEMRK9 - 31230 * 31231 * Input - 31232 * R1 - Address of 12 byte command 31233 * R10 - Address of our CMPBK 31234 * 31235 * Output - See Exit normal, Exit error 31236 * 31237 * Exit normal - 31238 * CC = 0, Table entry found 31239 * R7 = Address of JAMTABLE 31240 * 31241 * CC = 3, Table entry not found 31242 * 31243 * Exit error - 31244 * None 31245 * 31246 * General comments - 31247 * None 31248 * 31249 * End of specifications ***** </pre>	<pre> 20900001 20950001 21000001 21050001 21100001 21150001 21200001 21250001 21300001 21350001 21400001 21450001 21500001 21550001 21600001 21650001 21700001 21750001 21800001 21850001 21900001 21950001 22000001 22050001 22100001 22150001 22200001 22250001 22300001 22350001 22400001 </pre>
<pre> 31251 SRCHCMD HCPLENTNTR , 31722 HCPUSING CMPBK,R10 Addressability 31726 *-----* 31727 * Initialize: 31728 * SAVEMRK 31729 *-----* </pre>	<pre> 22500001 22600001 22700001 22750001 22800001 22850001 </pre>

Line **31251**: HCPLENTNTR default entry type is CALL.

Table 13. Annotated Listing for Sample CP Exit Routine 2 for CP Exit 1200 (continued)

Assembler Source					Commentary	
						Line 31734: A new CALL-type entry, HCPENTER or HCPLNTR, will be handed a new savearea. So even though the same labels are used in this subroutine, they are touching storage different from the caller's savearea.
0002F6 5870 A054	31731 *cmt	XC	SAVEWRK,SAVEWRK	HCP SVC actually does this	22950001	
0002FA 1277	31733 *			-----*	23050001	
0002FC A7D4 000C	31734 *			Start looking for a table entry that matches the command	23100001	
	31735 *			that R1 points to.	23150001	
	31736 *			-----*	23200001	
00054 31738	00054 31738	L	R7, CNPUSRF2	Address of first JAMTABLE	23300001	
00054 31739	31739	LTR	R7, R7	Any?	23350001	
00314 31740	00314 31740	BRMP	SCMDCC3	Addressability	23400001	
00054 31741	31741	HCPUSING	JAMTABLE, R7	-----*	23450001	
000300 00000 1000 00004	31745 SCMD100	DS	0H	Found it?	23550001	
000300 D50B 7004 000B	31746	CLC	JAMTCMD, 0(R1)	..yes	23600001	
000306 A784 000B	0031C 31747	BR	SCMDCC0	-----*	23650001	
00030A 5870 7000	00000 31749	L	R7, JAMTFWD	Address of the next JAMTABLE	23750001	
00030E 1277	31750	LTR	R7, R7	Any?	23800001	
000310 A724 FFF8	00300 31751	BRP	SCMD100	..yes	23850001	
000314	31753 SCMDCC3	DS	0H	Set condition code = 3	23950001	
000318 A7F4 0005	31754	CC	3	Return	24000001	
00031C	00322 31756	BRU	SCMDXIT	-----*	24050001	
00031C 5070 D034	31758 SCMDCC0	DS	0H	Return address of JAMTABLE	24150001	
	00034 31759	ST	R7, SAVER7	Set condition code = 0	24200001	
	31760	CC	0	Return	24250001	
	31762 *cmt	BRU	SCMDXIT	-----*	24300001	
	31764 *			-----*	24400001	
	31765 *			Return	24450001	
	31766 *			-----*	24500001	
000322	31768 SCMDXIT	DS	0H	-----*	24600001	
	31769	HCPLEXIT		-----*	24650001	
	32328	HCPDROP	R7		24750001	
	32330	HCPDROP	R10		24800001	
	32333 *	Start of specifications *****			24900001	
	32334 *	Entry Point Name - SRCHUID			24950001	
	32335 *	-----*			25000001	
	32336 *	Descriptive name -			25050001	
	32337 *	Search the chain of JAMTABLE control blocks for the			25100001	
	32338 *	first entry whose JAMTUID entry matches the argument			25150001	
	32339 *	passed in R1.			25200001	
	32340 *	Function -			25250001	
	32341 *	Table lookup			25300001	
	32342 *	-----*			25350001	
	32343 *	-----*			25400001	
	32344 *	-----*			25450001	

Table 13. Annotated Listing for Sample CP Exit Routine 2 for CP Exit 1200 (continued)

Assembler Source		Commentary
32345 *	Register usage -	25500001
32346 *	R0 -	25500001
32347 *	R1 - Address of 8 byte user ID	25600001
32348 *	R2 -	25650001
32349 *	R3 -	25700001
32350 *	R4 -	25750001
32351 *	R5 -	25800001
32352 *	R6 -	25850001
32353 *	R7 - Address of JAMTABLE	25900001
32354 *	R8 -	25950001
32355 *	R9 -	26000001
32356 *	R10 - Address of our CMPBK	26050001
32357 *	R11 - Dispatched VMOBK address	26100001
32358 *	R12 - Base register	26150001
32359 *	R13 - Save Area address	26200001
32360 *	R14 - Work register, linkage	26250001
32361 *	R15 - Work register, linkage	26300001
32362 *	SAVEMRK usage -	26350001
32363 *	SAVEMRK0 -	26400001
32364 *	SAVEMRK1 -	26450001
32365 *	SAVEMRK2 -	26500001
32366 *	SAVEMRK3 -	26550001
32367 *	SAVEMRK4 -	26600001
32368 *	SAVEMRK5 -	26650001
32369 *	SAVEMRK6 -	26700001
32370 *	SAVEMRK7 -	26750001
32371 *	SAVEMRK8 -	26800001
32372 *	SAVEMRK9 -	26850001
32373 *		26900001
32374 *	Input -	26950001
32375 *	R1 - Address of 8 byte user ID	27000001
32376 *	R10 - Address of our CMPBK	27050001
32377 *		27100001
32378 *	Output - See Exit normal, Exit error	27150001
32379 *	Exit normal -	27200001
32380 *	CC = 0, Table entry found	27250001
32381 *	R7 = Address of JAMTABLE	27300001
32382 *		27350001
32383 *		27400001
32384 *		27450001
32385 *	CC = 3, Table entry not found	27500001
32386 *	Exit error -	27550001
32387 *	None	27600001
32388 *		27650001
32389 *		27700001

Table 13. Annotated Listing for Sample CP Exit Routine 2 for CP Exit 1200 (continued)

Assembler Source		Commentary	
	32390 * General comments -	*	27750001
	32391 * None	*	27800001
	32392 *	*	27850001
	32393 * End of specifications *****	*	27900001
	32395 SRCHUID HCPLENTN ,		28000001
	32865 HCPUSING CMBPK,R10	Addressability	28050001
	32869 *		28150001
	32870 * Initialize:		28200001
	32871 * SAVEDWK	*	28250001
	32872 *	*	28300001
	32874 *cmt XC SAVEDWK,SAVEDWK	HCPUSVC actually does this	28400001
	32876 *		28500001
	32877 * Start looking for a table entry that matches the userid	*	28550001
	32878 * that R1 points to.	*	28600001
	32879 *	*	28650001
00033E 5870 A054	L R7,CMPUSRF2	Address of first JAMTABLE	28750001
000342 1277	LTR R7,R7	Any?	28800001
000344 A7D4 000C	BtNP SUIDCC3		28850001
	HCPUSING JAMTABLE,R7	Addressability	28900001
000348	DS 0H		29000001
000348 D507 7010 1000 00010	CLC JAMTUID,0(R1)	Found it?	29050001
00034E A784 000B	BtE SUIDCC0	..yes	29100001
000352 5870 7000	L R7,JAMTFWD	Address of the next JAMTABLE	29200001
000356 1277	LTR R7,R7	Any?	29250001
000358 A724 FFF8	BtP SUID100	..yes	29300001
00035C	DS 0H		29400001
000360 A7F4 0005	CC 3	Set condition code = 3	29450001
	BtU SUIDEXIT	Return	29500001
000364	DS 0H		29600001
000364 5070 D034	ST R7,SAVER7	Return address of JAMTABLE	29650001
	CC 0	Set condition code = 0	29700001
	BtU SUIDEXIT	Return	29750001
	32907 *		29850001
	32908 * Return		29900001
	32909 *	*	29950001
00036A	32911 SUIDEXIT DS 0H		30050001
	32912 HCPLEXIT ,		30100001
	33471 HCPDROP R7		30200001
	33473 HCPDROP R10		30250001

Table 13. Annotated Listing for Sample CP Exit Routine 2 for CP Exit 1200 (continued)

Assembler Source	Commentary
<pre> 33476 * Start of specifications ***** 33477 * Entry Point Name - JAMSAML 33478 * 33479 * Descriptive name - 33480 * Exit F200, load JAMTABLE 33481 * 33482 * Function - 33483 * Exit F200, load JAMTABLE 33484 * 33485 * Register usage - 33486 * R0 - Exit number 33487 * R1 - Scratch 33488 * R2 - Address of exit plist pointers 33489 * R3 - Argument to parser 33490 * R4 - Scratch 33491 * R5 - Argument to parser 33492 * R6 - Address of GSOBK 33493 * R7 - Argument to parser 33494 * R8 - Scratch 33495 * R9 - Address of DRBK 33496 * R10 - Scratch 33497 * R11 - Address of DRBK 33498 * R12 - Scratch 33499 * R13 - Address of JAMTABLE 33500 * R14 - Local linkage 33501 * R15 - Address of exit 1200 plist 33502 * R16 - Address of our CMPBK 33503 * R17 - Dispatched VMDBK address 33504 * R18 - Base register 33505 * R19 - Save Area address 33506 * R20 - Work register, linkage 33507 * R21 - Work register, linkage 33508 * 33509 * SAVEMRK usage - 33510 * SAVEMRK0 - Flags 33511 * SAVEMRK1 - Address of error message GSDBKs for operator 33512 * SAVEMRK2 - Address of DRBK 33513 * SAVEMRK3 - Address of GSOBK for parsing 33514 * SAVEMRK4 - Address of first JAMTABLE of new chain 33515 * SAVEMRK5 - Address of last JAMTABLE of new chain 33516 * SAVEMRK6 - 33517 * SAVEMRK7 - Address of empty JAMTABLE 33518 * SAVEMRK8 - 33519 * SAVEMRK9 - Error message number 33520 * </pre>	<p>Line 33476: This is local CP Exit F200, which loads a source CMS file into the system execution space. For an overview of local CP Exit F200's logic, see “Sample 2” on page 219 or see line 33570 below.</p>

Table 13. Annotated Listing for Sample CP Exit Routine 2 for CP Exit 1200 (continued)

Assembler Source		Commentary
33521 * Input -		Line 33524 : The registers passed to CP Exit F200 are: R0 - Exit number R1 - Address of exit 1200 plist addresses R2 - Address of XCRBK The registers passed to exit F200 are defined as they were for exit 1200 on line 22300. The only register that the mainline has control over is R1. Here, our exit 1200 routine is considered the mainline for exit F200. This code has passed on R1 as originally set by the mainline module for exit 1200, module HCPDIA.
33522 * R0 - Exit number		
33523 * R1 - Address of exit 1200 plist addresses		
33524 * R2 - Address of XCRBK		
33525 * Output - See Exit normal, Exit error		
33526 * Exit normal -		
33527 * JANTABLE blocks built		
33528 * Exit error -		
33529 * None		
33530 *		
33531 *		
33532 *		
33533 *		

Table 13. Annotated Listing for Sample CP Exit Routine 2 for CP Exit 1200 (continued)

Assembler Source	Commentary
<pre> 33534 * Serialization 33535 * Consider the possibility that we are running on a 33536 * 10-way. It is theoretically possible that 10 DIAL 33537 * commands could be running simultaneously, one on each 33538 * processor. Each calls exit 1200. Each exit 1200 calls 33539 * exit F200. One of the exit F200 tasks gets the CMPBK 33540 * lock exclusive, loads the JAMTABLE SOURCE file, and 33541 * then relinquishes the CMPBK lock. The second of the 33542 * original 10 tasks then does the same thing, then the 33543 * third, and so on. The JAMTABLE SOURCE file gets loaded 33544 * 10 times, needlessly. 33545 * 33546 * To avoid this unnecessary processing, the first task 33547 * will disable exit F200 before it relinquishes the CMPBK 33548 * lock. When this happens is immaterial. We simply 33549 * decided to do it early. When tasks 2 through 10 33550 * finally get a chance to run, they will notice that exit 33551 * F200 has become disabled, so they have nothing to do; 33552 * they simply return to exit 1200. 33553 * 33554 * In exit 1200, if an 11-th task comes along, it will 33555 * notice exit F200 is disabled and simply continue. It 33556 * will attempt to acquire the CMPBK lock shared, but will 33557 * be delayed until all tasks 1 through 10 relinquish the 33558 * CMPBK lock that they hold exclusive. When task 11 33559 * finally runs, the JAMTABLE SOURCE file will already 33560 * have been loaded. 33561 * </pre>	<p>Line 33534: Here is an explanation of our serialization.</p> <p>Consider the possibility that we are running on a 10-way. It is theoretically possible that 10 DIAL commands could be running simultaneously, one on each processor. Each calls exit 1200. Each exit 1200 calls exit F200. One of the exit F200 tasks gets the CMPBK lock exclusive, loads the JAMTABLE SOURCE file, and then relinquishes the CMPBK lock. The second of the original 10 tasks then does the same thing, then the third, and so on. The JAMTABLE SOURCE file gets loaded 10 times, needlessly.</p> <p>To avoid this unnecessary processing, the first task will disable exit F200 before it relinquishes the CMPBK lock. When this happens is immaterial. We simply decided to do it early. When tasks 2 through 10 finally get a chance to run, they will notice that exit F200 has become disabled, so they have nothing to do; they simply return to exit 1200.</p> <p>In exit 1200, if an 11th task comes along, it will notice exit F200 is disabled and simply continue. It will attempt to acquire the CMPBK lock shared, but will be delayed until all tasks 1 through 10 relinquish the CMPBK lock that they hold exclusive. When task 11 finally runs, the JAMTABLE SOURCE file will already have been loaded.</p>
<pre> 332500001 333000001 333500001 334000001 334500001 335000001 335500001 336000001 336500001 337000001 337500001 338000001 338500001 339000001 339500001 340000001 340500001 341000001 341500001 342000001 342500001 343000001 343500001 344000001 344500001 345000001 345500001 346000001 </pre>	<pre> * </pre>

Assembler Source

Line 33534 (continued):

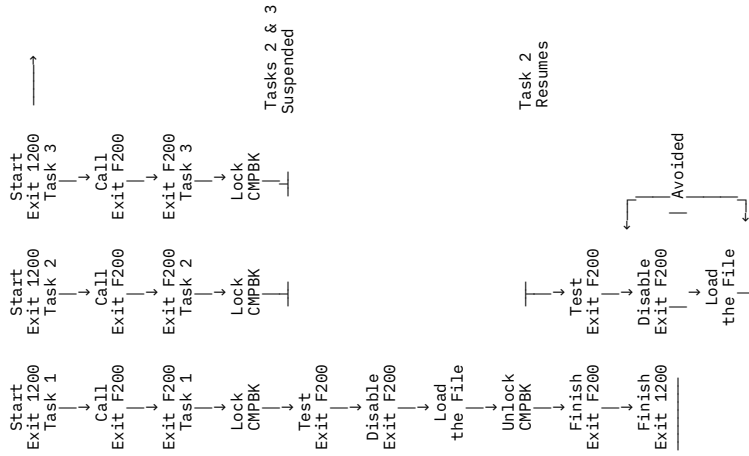


Table 13. Annotated Listing for Sample CP Exit Routine 2 for CP Exit 1200 (continued)

Assembler Source	Commentary
<pre>33562 * General comments - 33563 * We use standard CP message numbers, but the message 33564 * text comes from our own message repository. This is 33565 * indicated on the HCPCONSL MACRO by the COMPID= keyword. 33566 * 33567 * The parser will generate messages using the standard 33568 * 'HCP' message repository. 33569 *</pre>	<p>Line 33534 (continued):</p> <pre>graph TD Task3[Task 3 Resumes] --> U1[UnLock CMPBK] Task3 --> T[Test Exit F200] U1 --> F1[Finish Exit F200] F1 --> F2[Finish Exit 1200] T --> D[Disable Exit F200] D --> L[Load the File] L --> U2[UnLock CMPBK] U2 --> F3[Finish Exit F200] F3 --> F4[Finish Exit 1200] T -- Avoided --> U2</pre>
	<p>If the test for exit F200 were not performed, each task in turn would reload the file, which would be both unnecessary and wasteful. Moreover, this might seriously delay later tasks that, while still in exit 1200, see that exit F200 was disabled and try to drop right to the table searching.</p>
	<p>Line 33562: Something to keep in mind when using local message repositories. Standard CP services that you use may generate messages. If so, they will be from the message repository associated with 'HCP'.</p>

Table 13. Annotated Listing for Sample CP Exit Routine 2 for CP Exit 1200 (continued)

Assembler Source	Commentary
<pre> 33570 * Our logic for exit F200 33571 * validate input (specifically, exit number) 33572 * find CMPBK for component ID 'JAM' 33573 * if found 33574 * then do 33575 * acquire CMPBK lock exclusive 33576 * end 33577 * else do 33578 * allocate CMPBK, which also acquires CMPBK lock 33579 * end 33580 * test exit F200 33581 * if disabled 33582 * then exit 33583 * disable exit F200 33584 * get a DRBK for reading our table file 33585 * get a GSDBK for parsing 33586 * interconnect the DRBK and the GSDBK 33587 * open the DRBK 33588 * if failed 33589 * then exit 33590 * do until eof 33591 * read the next record 33592 * if sparse or all blanks or comment 33593 * then iterate 33594 * get a new JAMTABLE 33595 * call the parser 33596 * if failed 33597 * generate an error message 33598 * iterate 33599 * add the JAMTABLE to new chain 33600 * end 33601 * close the file 33602 * release the DRBK, GSDBK, unused JAMTABLE 33603 * if we had no errors, 33604 * then do 33605 * swap the new chain of JAMTABLEs for the old 33606 * release the old JAMTABLEs 33607 * end 33608 * else do 33609 * release the new JAMTABLEs 33610 * end 33611 * exit 33612 * 33613 * Note that "exit" always implies cleanup first. 33614 * 33615 * End of specifications ***** </pre>	<pre> 33500001 * 335100001 * 335150001 * 335200001 * 335250001 * 335300001 * 335350001 * 335400001 * 335450001 * 335500001 * 335600001 * 335650001 * 335700001 * 335750001 * 335800001 * 335850001 * 335900001 * 335950001 * 336000001 * 336050001 * 336100001 * 336150001 * 336200001 * 336250001 * 336300001 * 336350001 * 336400001 * 336450001 * 336500001 * 336550001 * 336600001 * 336650001 * 336700001 * 336750001 * 336800001 * 336850001 * 336900001 * 336950001 * 337000001 * 337050001 * 337100001 * 337150001 * 337200001 * 337250001 * 337300001 * </pre>
<pre> 33617 JAMSAMD HCPENTER CALL,SAVE=DYNAMIC 34203 *-----* 34204 * Initialize: 34205 * SAVEDWK 34206 *-----* 34208 *cmt XC SAVEDWK,SAVEDWK HCPSPVC actually does this 34236 *-----* 34237 * Test exit number; leave if not F200. 34238 *-----* </pre>	<pre> 37400001 37500001 37550001 37600001 37650001 37750001 39150001 39200001 39250001 </pre>

Table 13. Annotated Listing for Sample CP Exit Routine 2 for CP Exit 1200 (continued)

Assembler Source		Commentary	
00038A 5900 C0A0			<p>Line 34240: A little defensive programming here. You could code the same entry point for more than one exit. Whether you do or not, it would be good practice to verify that R0 contains the exit number that you expect.</p>
00038E A774 01AE			
000392 1891			
000A0 34240	C	R0,=A(XIT@F200)	
006EA 34241	BRNE	LDEXIT	
34243	LR	R9,R1	
34244	HCPUSING	X1200,R9	
34248	*	-----*	
34249	*	* Get our component ID block.	
34250	*	* If our component ID block does not exist (which is	
34251	*	* indicated by RU5 = 4 from FIND), then we need to	
34252	*	* allocate it. Allocate will also acquire the CMPBK	
34253	*	* lock exclusive.	
34254	*	-----*	
34256	HCPXSERV	FIND, COMPID= 'JAM'	<p>Line 34256: Here is how to specify your component ID when finding, allocating, deallocating, locking, or unlocking. There will be at most one component ID block (CMPBK) for a specific component. The component ID is 3 characters long. You can specify your component ID in any of these ways:</p> <pre> HCPXSERV xxxx,COMPID= 'JAM' LA Rx,DCJAM HCPXSERV xxxx,COMPID=(Rx) R1 recommended HCPXSERV xxxx,COMPID=DCJAM DCJAM DC C 'JAM' </pre>
35838	HCPCASRC	(LD100, LD033, LDEXIT), ELSE=LDEXIT	
		00: found	
		04: not found	
		08: bad argument passed	
		?: invalid rc	

Table 13. Annotated Listing for Sample CP Exit Routine 2 for CP Exit 1200 (continued)

Assembler Source		Commentary	
0003C0	35861 LD033 DS HCPXSERV ALLOCATE, COMPID= 'JAM'	0H	40350001
	35862		40400001
	37447 HCPCASRC (LD066, LD100, LDEXIT)	00: success, lock obtained 04: exists, not locked 08: passed bad R1 12: passed bad R2 ?: invalid IC	X40450001 X40500001 X40550001 X40600001 40650001
0003EE 19A1 0003EE 13F4 0019 0003F0 A7F4	37470 LD066 DS LR BTU	0H R10, R1 LD166	40750001 40800001 40850001
	37471		
	00422 37472	Address of our CMPBK	
<p>Line 35861: Before using your component ID block, you must make a request to have it created.</p> <p>If the CMPBK was allocated by this HCPXSERV ALLOCATE:</p> <ul style="list-style-type: none">• R15 returned to you will contain 0.• You will have the CMPBK lock exclusive; you will need to UNLOCKEXCLUSIVE later. The presumption is that you intend to initialize the control block and that you do not want any interference.• The CMPBK will be initialized to binary zeros.• R1 returned to you will contain the address of the CMPBK. <p>If the CMPBK was not allocated by this HCPXSERV ALLOCATE:</p> <ul style="list-style-type: none">• R15 returned to you will contain 4 (assuming all arguments were valid).• You will not have the CMPBK lock.• R1 returned to you will contain the address of the CMPBK.			

Table 13. Annotated Listing for Sample CP Exit Routine 2 for CP Exit 1200 (continued)

Assembler Source		Commentary	
0003F4	37474 LD100 DS 0H 37475 HCPXSERV LOCKEXCLUSIVE,COMPID='JAM'	40950001 41000001	Line 37474 : If the HCPXSERV ALLOCATE did not create a new CMPBK because it already existed, then exclusive control over the CMPBK was not granted. We need to get exclusive control before proceeding.
	39057 HCPCASRC (LD133, LD033, LDEXIT, LD033, LDEXIT), ELSE=LDEXIT	X41050001 X41100001 X41150001 X41200001 X41250001 41300001	
000420 000420 18A1	39084 LD133 DS 0H 39085 LR R10,R1	41400001 41450001	
000422 000422 9601 D058	39087 LD166 DS 0H 39088 OI SAVEWRK0,X'01' 39089 HCPUSING CMPBK,R10	41550001 41600001 41650001	
	39093 *-----* 39094 * Before we go any further, let's see if exit F200 has 39095 * become disabled. If so, we will assume it is because 39096 * another instance of this exit has already done what we 39097 * intended to do. Therefore, we can simply return and 39098 * expect that the JAMTABLE data structure has been built. 39099 *-----*	41750001 41800001 41850001 41900001 41950001 42000001 42050001	
	39101 HCPXSERV TEST,EXIT=F200, If exit F200 disabled, quit DISABLED=L0700	X42150001 42200001	Line 39101 : This is how any exit can be tested for enabled or disabled.
			<ul style="list-style-type: none"> • If enabled, execution continues at the next instruction. • If disabled, execution continues at the 'DISABLED=' label.
			'EXIT=' could be specified in a register, as in 'EXIT=(Rx)'.
	39114 HCPXSERV DISABLE,EXIT=F200 Disable it for next task	42300001	Line 39114 : This is how any exit can be disabled. 'EXIT=' could be specified in a register, as in 'EXIT=(Rx)'.
	40724 *-----* 40725 * Get a DRBK, save its address in SAVEWRK2. 40726 * Fill in the DRBK. 40727 *-----*	42400001 42450001 42500001 42550001	Line 40724 : The DRBK is the control block used in all requests to read a CMS file. It is analogous to the CMS control block FSCB.

Table 13. Annotated Listing for Sample CP Exit Routine 2 for CP Exit 1200 (continued)

Assembler Source		Commentary	
000476 5810 C0B0	40729	HCPGETST ID=DRBK	Get a Data Request Block
00047A 5810 C0B0	41155+	L R1,=F'1488'	CBI offset
00047A 58F0 0950	41420+	DC 0A14(HCPFREE)	Force a cross reference
	41974+	L R15,PFxFREE	
	42159+*cmt	MVI PFxiACE,SACPRIM/256	Caller module's Tmode
	42160+*cmt	MVI PFxiACE,SACPRIM/256	Callee entry's Tmode
00047E 0CEF	42347+	SACF SACPRIM	Enter needed Tmode
	42347+	BASSM R14,R15	Static linkage, set Amode
	42534+*cmt	SACF SACPRIM	Restore caller's Tmode
000480 1841	42720	LR R4,R1	Address of DRBK
	42721	HCPUSING DRBK,R4	Addressability
	42721	ST R4,SAVEWRK2	Save the address
000482 5040 D060	00060 42724		
000486 D207 4028 C078 00028 00078	42726	MVC DRBFIDFN'=CL(L'DRBFIDFN)'JAMTABLE'	fname
00048C D207 4030 C080 00030 00080	42727	MVC DRBFIDFT'=CL(L'DRBFIDFT)'SOURCE'	itype
000492 D201 407C C102 0007C 00102	42728	MVC DRBACSBX'=FL(L'DRBACSBX)'-1'	fmode = 'x'
	42729 *cmt	MVC DRBRECONO'=FL(L'DRBRECONO)'0'	
000498 D203 406C C0B4 0006C 000B4	42731	MVC DRBBUFsz'=FL(L'DRBBUFsz)'4000'	
	42732	CKMAINT 4000,GT,FREMX*8-GSDHLEN	
	42737 *	*****	*****
	42738 *	* Get a maximum size GSDBK, save its address in SAVEWRK3.	*
	42739 *	*****	*****
0004AC 1831	42741	HCPGETST ID=GSDBK,LEN=FREMX	Get a big GSDBK
	44309	LR R3,R1	Address of GSDBK
	44310	HCPUSING GSDBK,R3	Addressability
0004AE 5030 D064	44313	ST R3,SAVEWRK3	Save the address
0004B2 D201 300A C104 0000A 00104	44315	MVC GSDFRESZ'=AL(L'GSDFRESZ)'(FREMX)	size
	44317 *	*****	*****
	44318 *	* Interconnect the DRBK and the GSDBK.	*
	44319 *	* We will setup DRBBUFAD to point to GSDDATA so that when	*
	44320 *	* we read a record it will be placed right into the GSDBK,	*
	44321 *	* almost ready for parsing.	*
	44322 *	*****	*****
0004B8 4110 3010	00010 44324	LA R1,GSDDATA	
0004BC 5010 4068	00068 44325	ST R1,DRBBUFAD	DRBBUFAD --> GSDDATA
	44327 *	*****	*****
	44328 *	* Open the DRBK.	*
	44329 *	*****	*****
0004C0 4110 4000	00000 44331	LA R1,DRBK	Open the DRBK
	44332	HPCALL HCPZIOPN	
0004CE D501 407E C106 0007E	00106 45909	CLC DRBRETCO,=AL(L'DRBRETCO)'(DRBOK) Success?	
0004D4 A774 008E	005F0 45910	BtRNE LD433 ,	..no
	45912 *	*****	*****
	45913 *	* Read the file until we reach e-o-f.	*
	45914 *	*****	*****
0004D8 5830 D064	00064 45916	DS 0H	
0004D8 5830 D064	00064 45917	L R3,SAVEWRK3	Fix R3 before we crash
0004DC 5840 D060	00060 45918	L R4,SAVEWRK2	Fix R4 before we crash

Line 44329: Here is an expansion of a HCPGETST macro request. The expansion includes comment statements that might not be comments if a different set of module and entry point attributes were assigned to the caller and to the callee. The MVI instructions might actually save values other than SACPRIM. The SACF instructions might set PSW translations mode to something other than Primary Space mode. Linkage services depend on accurate attribute definitions.

Line 44327: This is how you open a CMS file for reading.

After you open the file, check the return code in DRBRETCO. The return codes are defined in HCPDRBK COPY.

Table 13. Annotated Listing for Sample CP Exit Routine 2 for CP Exit 1200 (continued)

Assembler Source		Commentary	
0004E0 4100 0001	00001 45919	LA R0,1	Increment to next record
0004E4 5A00 405C	0005C 45920	A R0,DRBRECNO	*
0004E8 5000 405C	0005C 45921	ST R0,DRBRECNO	*
0004EC 4110 4000	00000 45923	LA R1,DRBK	Read a record
	00000 45924	HCPCALL HCPZIGET	*
0004FA D501 407E	00108 47501	CLC DRBRETCD,=AL(L'DRBRETCD) (DRBEOF) e-o-f?	
000500 A784 0095	0062A 47502	BtE LD533	..yes
000504 D501 407E	00106 47503	CLC DRBRETCD,=AL(L'DRBRETCD) (DRBOK) Success?	
00050A A774 006D	005E4 47504	BtNE LD400	..no
	47506 *	-----*-----	
	47507 *	* If the record was empty, then iterate.	
	47508 *	* If the record is a comment, then iterate.	
	47509 *	-----*-----	
00050E 9120 4055	00055	TM DRBFLAG2,DRBSPARS	Sparse file?
000512 A714 FFE3	004D8 47511	BtO LD200	..yes, try the next record
000516 D503 4070	0006C 47514	CLC DRBRECSZ,DRBBUFSZ	Bigger than buffer?
00051C A724 005E	005D8 47515	BtH LD366	..yes, error
000520 5800 4068	00068 47517	L R0,DRBBUFAD	A(data)
000524 5810 4070	00070 47518	L R1,DRBRECSZ	L'data
	00070 47519	R14,0	Immaterial
000528 58F0 C0C8	000C8 47520	L R15,=AL1(C' ',0,0,0)	(pad=c', length=0)
00052C 0F0E	000C8 47521	CLCL R0,R14	All blanks?
000532 E784 FFD5	004D8 47522	BtE LD200	..yes, try the next record
000532 955C 3010	00010	CLI GSDDATA,C'*	Comment?
000536 A784 FFD1	004D8 47525	BtE LD200	..yes, try the next record

Line **45919**: This is how to read the next record in a CMS file.

All I/O to a CMS file is by record number; there is no "get next" request. There is no request for more than a single record.

CMS file I/O is not intended to be a high performer, because it is expected that CP will not be reading large quantities of CMS data.

Check the return code in DRBRETCD afterward.

Forget about RECFM. Whether the file is F or V, you should always use DRBRECSZ as the size of the record that was read. Do not require any particular RECFM, because it just adds burden to the person creating the file.

Table 13. Annotated Listing for Sample CP Exit Routine 2 for CP Exit 1200 (continued)

Assembler Source		Commentary	
00053A 5800 4070	00070 47527	L R0,DRBRECZ	46850001
00053E 4000 300E	0000E 47528	STH R0,GSDDCNT *	46900001
000542 4100 0000	00000 47529	LA R0,0	46950001
000546 4000 300C	0000C 47530	STH R0,GSDSCAN *	47000001
	47532 *	-----*	47100001
	47533 *	* If we have a ready JAMTABLE block, use it.	47150001
	47534 *	* If we don't, then allocate one.	47200001
	47535 *	-----*	47250001
00054A 5870 D074	00074 47537	L R7,SAVEWRK7	47350001
00054E 1277 0000	00038 47538	LTR R7,R7	47400001
000550 4724 000C	00568 47539	BRP LD233	47450001
		Address of JAMTABLE, or zero	
		Any?	
		..Yes, use it	
000562 1871	47541	HCPGETST LEN=0+(JAMTABLN*7)/8	47550001
000564 5070 D074	49108	LR R7,R1	47600001
	00074 49109	ST R7,SAVEWRK7	47650001
		Save address of JAMTABLE	
000568	49111 LD233	DS 00H	47750001
	49112	HCPUSING JAMTABLE,R7	47800001
	49116 *	-----*	47900001
	49117 *	* Call the parser.	47950001
	49118 *	-----*	48000001
	49119 *	* Part of what the parser does is to clear the primary	48050001
	49120 *	* plist to binary zeros.	48100001
	49121 *	-----*	48150001
	49122 *	* By passing R3 with the address of the HCPCFDEF	48200001
	49123 *	* that we want, we are actually pretending that we parsed	48250001
	49124 *	* off the first token and used it to find the HCPCFDEF that	48300001
	49125 *	* we want.	48350001
	49126 *	-----*	48400001
	49127 *	* If the parser detected a syntax error, then we will queue	48450001
	49128 *	* the error GSDBK that the parser generated and a GSDBK to	48500001
	49129 *	* tell the operator what record was bad.	48550001
	49130 *	* Loop for another record.	48600001
	49131 *	-----*	48650001
	49132 *	* We will use X1200TRT as a scratch area in order to leave	48700001
	49133 *	* X1200UMD alone. We anticipate that uses of X1200TRT by	48750001
	49134 *	* other exit routines would be transitory in the sense that	48800001
	49135 *	* such routines would initialize the TRT area before use.	48850001
	49136 *	* The user words area (X1200UMD) are documented as a	48900001
	49137 *	* suggested area for exit routines to pass data to each	48950001
	49138 *	* other. If we avoid the user words, we won't tromp on any	49000001
	49139 *	* such sharing.	49050001
	49140 *	-----*	49100001
000568 5820 9008	00008 49142	L R2,X1200TRT	49200001
00056C 41F0 0000	00000 49143	LA R15,PFXPGB	49250001
000570 50F0 2000	00000 49144	ST R15,0(R2)	49300001
		Additional bases	
		An additional base	
		*	
000574 4100 C7D8	007D8 49146	LA R0,SYNTAX	49400001
000578 4110 7000	00000 49147	LA R1,JAMTABLE	49450001
	*cmt	L R2,X1200TRT	49500001
00057C 4130 C83D	0083D 49149	LA R3,CF	49550001
000580 5840 D064	00064 49150	L R4,SAVEWRK3	49600001

Table 13. Annotated Listing for Sample CP Exit Routine 2 for CP Exit 1200 (continued)

Assembler Source		Commentary	
00058E 5830 D064	49151	HPCALL HCPZPRPG	Line 49151 : Parsing the contents of a GSDBK. We are passing the following registers to parser entry point HCPZPRPG: <ul style="list-style-type: none"> • R0 - Contains the address of the HCPDOSYN macro. • R1 - Contains the address of the primary PLIST, as specified by the 'PL=' parameter of the HCPDOSYN macro. • R2 - Contains either: Zero The address of the 'BASES=' PLIST, as specified on the HCPDOSYN macro. • R3 - Contains either: Zero The address of the HCPCFDEF macro. • R4 - Contains the address of the GSDBK that we are parsing. On return, the following registers contain information: <ul style="list-style-type: none"> • R0 - Contains either: Zero, if there were no syntax errors A nonzero return code, if there were syntax errors. • R15 - Contains a return code of: Zero, if there were no syntax errors Nonzero, if there were syntax errors. Error messages will have been written unless RETGSDBK=xxx was specified on the HCPDOSYN macro. Refer to line 70517 below.
000592 5840 D060	50727	L R3,SAVEWRK3	
000596 12FF	50728	L R4,SAVEWRK2	
000598 A784 0034	50729	LTR R15,R15	
	50730	BIZ LD466	
	50732	*-----*	Parse the GSDBK line Fix R3 before we crash Fix R4 before we crash Check out return code ..success *-----* * Bad syntax detected by the parser. * Connect the error message GSDBK to any existing ones. *-----*
	50733		
	50734		
	50735		
			49650001
			49700001
			49750001
			49800001
			49850001
			49950001
			50000001
			50050001
			50100001

Table 13. Annotated Listing for Sample CP Exit Routine 2 for CP Exit 1200 (continued)

Assembler Source					Commentary	
00059C	5810	7000	00000	50737	R1, JAMTGSD	Address of error GSDBK
0005A0	5830	D05C	0005C	50739	L R3, SAVEWRK1	Address of first, or zero
0005A4	D503	D05C	0A00	50740	CLC SAVEWRK1, PFX0	A chain to search?
0005AA	A774	0006	005B6	50741	BrNE LD266	..yes, go search it
0005AE	5910	D05C	0005C	50743	ST R1, SAVEWRK1	Address of the next
0005B2	A7F4	000D	005CC	50744	BrU LD333	Say which record is bad
0005B6	D503	3000	0A00	50746	DS 0H	Another follows?
0005B8	A784	0006	005C8	50748	CLC GSDNEXT, PFX0	..no, add this one
0005C0	5830	3000	00000	50749	BrE LD300	Address of the next
0005C4	A7F4	FFF9	005B6	50750	BrU LD266	Continue looking
0005C8						
0005C8	5010	3000	00000	50752	DS 0H	Address of the next
				50753	ST R1, GSDNEXT	
				50755	*	-----*
				50756	* Add an error message GSDBK that indicates which record	*
				50757	* failed parsing.	*
				50758	* Go back and read the next record.	*
				50759	*	-----*
0005CC	5800	C0D0	00000	50761	DS 0H	Error message number
0005D0	A785	0093	006F6	50762	L R0, =A(MS670001)	Format the GSDBK
0005D4	A7F4	FF82	004D8	50763	BrAS R8, LDERR	Go read another record
				50764	BrU LD200	
				50766	*	-----*
				50767	* Add an error message GSDBK that indicates that the	*
				50768	* record is too long.	*
				50769	* Go back and read the next record.	*
				50770	*	-----*
0005D8	5800	C0D4	00004	50771	DS 0H	Error message number
0005DC	A785	008D	006F6	50772	L R0, =A(MS670702)	Format the GSDBK
0005E0	A7F4	FF7C	004D8	50773	BrAS R8, LDERR	Go read another record
				50774	BrU LD200	
				50776	*	-----*
				50777	* Add an error message GSDBK that indicates that we had	*
				50778	* an I/O error.	*
				50779	* Go back and read the next record.	*
				50780	*	-----*

Line 50737: Remember the HCPDOSYN keyword 'RETGSDBK=JAMTGSD'?

Table 13. Annotated Listing for Sample CP Exit Routine 2 for CP Exit 1200 (continued)

Assembler Source		Commentary	
0005E4	50782 LD400	DS	0H
0005E4	50783 L	R0,=A(NS670201)	Error message number
0005E8	50784 BRAS	R8,LDERR	Format the GSOBK
0005EC	50785 BRU	LD200	Go read another record
	50787 *	*****	*****
	50788 *	Add an error message GSOBK that indicates that we cannot	*
	50789 *	find the file.	*
	50790 *	Quit altogether.	*
	50791 *	*****	*****
0005F0	50793 LD433	DS	0H
0005F0	50794 L	R0,=A(NS670301)	Error message number
0005F4	50795 BRAS	R8,LDERR	Format the GSOBK
0005F8	50796 BRAS	R8,LDUNLOCK	Unblock our CMPBK
0005FC	50797 BRU	LD633	Cleanup and return
	50799 *	*****	*****
	50800 *	Connect the new JAMTABLE to our new chain of JAMTABLES.	*
	50801 *	*****	*****
000600	50803 LD466	DS	0H
000600	50804 L	R1,SAVEWRK5	Current last
000604	50805 LTR	R1,R1	Any here?
000606	50806 BRNP	LD500	...no, this is the first
00060A	50808 ST	R7,JAMTFWD-JAMTABLE(R1)	No empty JAMTABLE
00060E	50809 SLR	R7,R7	Clear away here, too
000610	50810 ST	R7,SAVEWRK7	Go read another record
00614	50811 BRU	LD200	
000618	50813 LD500	DS	0H
000618	50814 ST	R7,SAVEWRK4	Remember first new JAMTABLE
00061C	50815 ST	R7,SAVEWRK5	Remember last new JAMTABLE
000620	50816 SLR	R7,R7	No empty JAMTABLE
000622	50817 ST	R7,SAVEWRK7	Clear away here, too
000626	50818 BRU	LD200	Go read another record
	50820 *	*****	*****
	50821 *	E-o-f.	*
	50822 *	Close the file.	*
	50823 *	*****	*****
00062A	50825 LD533	DS	0H
00062A	50826 LA	R1,DRBK	Close the DRBK
	50827 HPCALL	HCPZICLS	

Line **50820**: Always close the CMS file before you release the DRBK.

Table 13. Annotated Listing for Sample CP Exit Routine 2 for CP Exit 1200 (continued)

Assembler Source		Commentary	
52404	*-----*		54800001
52405	* If we had no error,		54850001
52406	* then install our new chain of JAMTABLES		54900001
52407	* else free our new chain of JAMTABLES		54950001
52408	*-----*		55000001
52409	* What we do is cause SAVEWRK4 to point to whatever is to		55050001
52410	* be freed, whether this is the old chain of JAMTABLES that		55100001
52411	* we are replacing, or the new chain that we were building.		55150001
52412	*-----*		55200001
000638	D503 D05C 0A00 0005C	CLC SAVEWRK1,PFx0	55300001
00063E	A774 0009	BIWE LD566	55350001
000642	5800 D068	L R0,SAVEWRK4	55450001
000646	D203 D068 A054 00068	MVC SAVEWRK4,CMPUSRF2	55500001
00064C	5800 A054	ST R0,CMPUSRF2	55550001
000650		DS 0H	55650001
000650	5810 D068	L R1,SAVEWRK4	55700001
000654	1211	LTR R1,R1	55750001
000656	A7D4 000C	BrNP LD600	55800001
00065A	D203 D068 1000 00068	MVC SAVEWRK4,JAMTFWD-JAMTABLE(R1)	55850001
		HCPRELIST BLOCK=(R1)	55900001
00066A	A7F4 FFF3	BrU LD566	55950001
00066E		DS 0H	56050001
00066E	A785 0099	BRAS R8,LDUNLOCK	56100001
53999	*-----*		56200001
54000	* If we have any error message GSDBKs,		56250001
54001	* then send them on to the operator		56300001
54002	*-----*		56350001
54004	LD633	DS 0H	56450001
000672	5830 D05C	L R3,SAVEWRK1	56500001
000676	1233	LTR R3,R3	56550001
000678	A7D4 0019	BrNP LD666	56600001
00067C	D203 D05C 3000 0005C	MVC SAVEWRK1,GSDNEXT	56650001
000682	4140 3010	LA R4,GSDDATA	56700001
000686	4820 300E	LH R2,GSDDCNT	56750001
		HCPCONS1 WRITE,	X56800001
		DATA=((R4),(R2)),	X56850001
		DATATYPE=FULLMSG,	X56900001
		DESTINATION='OPERATOR'	56950001
55671		HCPRELIST ID=GSDBK,BLOCK=(R3)	57050001
0006A6	A7F4 FFE6	BrU LD633	57150001
0006AA		DS 0H	57250001

Table 13. Annotated Listing for Sample CP Exit Routine 2 for CP Exit 1200 (continued)

Assembler Source		Commentary	
0006AA	5810 D060	57246 *-----*	57350001
0006AA	1211	57247 * Cleanup and return.	57400001
0006AA	A7D4 0007	57248 *-----*	57450001
0006AA	A785 007B	57250 LD700 DS 0H	57550001
		57251 BIAS R8,LDUNLOCK UnLock our CMPBK	57600001
		57253 *-----*	57700001
		57254 * Release the DRBK, the GSDBK, any unused JAMTABLE.	57750001
		57255 *-----*	57800001
0006AE	5810 D060	57257 L R1,SAVEWRK2 Address of DRBK, if any	57900001
0006B2	1211	57258 LTR R1,R1	57950001
0006B4	A7D4 0007	57259 BtNP LD733	58000001
		57260 HCPRELST ID=DRBK,BLOCK=(R1)	58050001
0006C2	5810 D064	58829 LD733 DS 0H	58100001
0006C6	1211	58830 L R1,SAVEWRK3 Address of GSDBK, if any	58150001
0006C8	A7D4 0007	58831 LTR R1,R1	58200001
		58832 BtNP LD766	58250001
		58833 HCPRELST ID=GSDBK,BLOCK=(R1)	58300001
0006D6		60402 LD766 DS 0H	58350001
0006D6	5810 D074	60404 L R1,SAVEWRK7 Address of JAMTABLE, if any	58450001
0006DA	1211	60405 LTR R1,R1	58500001
0006DC	A7D4 0007	60406 BtNP LD800	58550001
		60407 HCPRELST BLOCK=(R1)	58600001
0006EA		61975 LD800 DS 0H	58650001
		61977 *-----*	58750001
		61978 * Return.	58800001
		61979 *-----*	58850001
0006EA	D203 D054 0A00 00054 00A00 61982	61981 LDEXIT DS 0H	58950001
0006EA	D203 D054 0A00 00054 00A00 61982	61982 MVC SAVER15,PFX0 Rc <- 0	59000001
		61984 HCPEXIT EP=(JAMSAMLD),SETCC=NO	59100001
	0F200	62542 XIT0F200 EQU X'F200'	59200001

Table 13. Annotated Listing for Sample CP Exit Routine 2 for CP Exit 1200 (continued)

Assembler Source		Commentary	
0006F6	5000 D07C	62544 * ----- message GSDBK that includes the record number *	59300001
0006F6	5000 D07C	62545 * in error.	59350001
0006FA	5830 D064	62546 * The substitution text that we build in GSDDATA will look *	59400001
0006FE	5840 D060	62547 * Like this, as if we had assembled these DCs. *	59450001
		62548 *	59500001
		62549 * DC C'fname',X'00'	59550001
		62550 * DC C'ftype',X'00'	59600001
		62551 * DC C'fmode',X'00'	59650001
		62552 * DC C'recnumber',X'FF'	59700001
		62553 *	59750001
		62554 *	59800001
		62555 * ----- *	59850001
0006F6	5000 D07C	62557 LDERR DS 0H	59950001
0006F6	5000 D07C	62558 ST R0,SAVEWRK9	60000001
0006FA	5830 D064	62559 L R3,SAVEWRK3	60050001
0006FE	5840 D060	62560 L R4,SAVEWRK2	60100001
000702	4150 3010	62562 LA R5,GSDDATA	60200001
000706	4100 3010	62563 LA R0,GSDDATA	60250001
00070A	5810 C0E4	62564 L R1,=F'4000'	60300001
		62565 *cmt LA R14,0	60350001
00070E	58F0 C0E8	62566 L R15,=AL1(0,0,0,0)	60400001
000712	0E0E	62567 MVCL R0,R14	60450001
000714	D207 5000 4028	62568 MVC 0(L'DRBFIDFN,R5),DRBFIDFN	60500001
00071A	4150 5009	62569 LA R5,1+L DRBFIDFN(R5)	60550001
00071E	D207 5000 4030	62571 MVC 0(L'DRBFIDFT,R5),DRBFIDFT	60650001
000724	4150 5009	62572 LA R5,1+L DRBFIDFT(R5)	60700001
000728	4810 407C	62574 LH R1,DRBACSBX	60800001
		62575 HPCALL HCPCVUBM *	60850001
000732	18F0	64094 LR R15,R0	60900001
000734	06F0	64095 BCTR R15,0	60950001
000736	44F0 C79A	64096 EX R15,LDWVC	61000001
00073A	415F 5002	64097 LA R5,1+1(R15,R5)	61050001
00073E	5810 405C	64098 L R1,DRBRECNO	61100001
		64099 HPCALL HCPCVTRM *	61150001
000748	18F0	65639 LR R15,R0	61200001
00074A	06F0	65640 BCTR R15,0	61250001
00074C	44F0 C79A	65641 EX R15,LDWVC	61300001
000750	415F 5002	65642 LA R5,1+1(R15,R5)	61350001
000754	0650	65644 BCTR R5,0	61450001
000756	92FF 5000	65645 MVI 0(R5),X'FF'	61500001
00075A	5850 D07C	65646 L R3,SAVEWRK9	61550001

Table 13. Annotated Listing for Sample CP Exit Routine 2 for CP Exit 1200 (continued)

Assembler Source		Commentary	
65647	HCPCONSL WRITE, COMPID='JAM', DATATYPE=FULLEMSG, DESTINATION=GSDBK, REPOSNUM=(R5), RETSDBKADDR=(R1), SUBDATA=GSDDATA	X61600001 X61650001 X61700001 X61750001 X61800001 X61850001 61900001	Line 65647 : Writing messages from a local message repository.
67257 67258 67259	*-----* * Connect the error message GSDBK to any existing ones. * *-----*	62000001 62050001 62100001	Here is how to specify your component ID for your own message repository:
00076E 5830 D05C 000778 D503 D05C 00077A A774 0005	0005C 0A00 0005C 00A00 0005C 00782 67262	HCPCONSL xxxx,COMPID='JAM' LA Rx,DCJAM HCPCONSL xxxx,COMPID=(Rx) R1 not allowed HCPCONSL xxxx,COMPID=DCJAM	
00077C 5010 D05C 000780 07F8	0005C 0A00 0005C 00782 67266	DCJAM DC C'JAM'	
000782 D503 3000 000788 A784 0006 00078C 5830 3000 000790 A7F4 FFF9	0005C 0A00 0005C 00782 67269 00794 67270 00080 67271 00782 67272	Here is how your messages are connected to HCPCONSL:	
000794 5010 3000 000798 07F8	0005C 0A00 0005C 00794 67275 00798 67276	• The COMPID= component ID is associated to your message repositories with the ASSOCIATE MESSAGE command or configuration file statement.	
00079A D200 5000	0005C 0A00 0005C 0079A 67278	• Your message repositories are created with XEDIT.	
0007A0 9103 D058 0007A4 0788	0005C 0A00 0005C 007A4 67292 007A4 67293	• Your message repositories are compiled with the CMS GENMSG command.	
0007A6 9102 D058 0007AA A714 000C	0005C 0A00 0005C 007A6 67295 007AA 67296	• Your message repositories are load with the CPXLOAD command or configuration file statement.	

Table 13. Annotated Listing for Sample CP Exit Routine 2 for CP Exit 1200 (continued)

Assembler Source		Commentary	
0007BC 94FC D058 0007C0 07F8	00058	67298	HCPXSERV UNLOCKEXCLUSIVE ,COMPID='JAM'
		68881	NI SAVEWRK0,X'FC'
		68882	BR R3 CMPBK no longer locked Return
0007C2		68884	LDUN200 DS 0H UnLock our CMPBK
0007D0 94FC D058 0007D4 07F8	00058	70468	NI SAVEWRK0,X'FC'
		70469	BR R3 CMPBK no longer locked Return
		70471	HCPDR0P R3
		70473	HCPDR0P R4
		70475	HCPDR0P R7
		70477	HCPDR0P R9
		70479	HCPDR0P R10
		64050001	Line 67298: Always relinquish control over your CMPBK with the complementary "unlock" request. To do otherwise invites disaster.
		64150001	
		64200001	
		64300001	
		64450001	
		64500001	
		64600001	
		64650001	
		64700001	
		64750001	
		64800001	

Table 13. Annotated Listing for Sample CP Exit Routine 2 for CP Exit 1200 (continued)

Assembler Source	Commentary
<pre> 70482 *-----* 70483 * Syntax definition of the source file for JAMTABLE. 70484 * The parser doesn't look at this HCPCFDEF definition 70485 * because we tell it that we have already used a token 70486 * (really, we pretended to) and found the desired HCPCFDEF 70487 * (there could have been many). We tell the parser all of 70488 * this by passing the address of this HCPCFDEF in R3 when 70489 * we call the parser. 70490 *-----* </pre>	<p>Line 70482: Defining syntax to the CP parser.</p>
<pre> 70492 CF HCPCFDEF 'JAMTABLE',PROCESS=NONE 70493 * 70494 * HCPSTDEF REQMATCH=YES The command 70495 * HCPTKDEF TYPE=TOKEN, * ERROR=MS670635, * STORE=JAMTCMD 70496 * 70497 * HCPSTDEF REQMATCH=YES, * MISSING=MS002001 70498 * HCPTKDEF TYPE=USERID, * ERROR=MS000701, * STORE=JAMTUID </pre>	<pre> 64900001 64950001 65000001 65050001 65100001 65150001 65200001 65250001 65300001 </pre>
<pre> 70499 * HCPSTDEF REQMATCH=YES, The target VDEV number 70500 * MISSING=MS002201 70501 * HCPTKDEF '*,', C '*' implies F'-1' to us * SOURCE=PEXPG(PFXEFS), * STORE=JAMTVDN 70503 * HCPTKDEF '=,', C '=' is special to us * SOURCE=(C=' ',0),LEN=1, * STORE=JAMTVDN 70505 * HCPTKDEF TYPE=HEX, * ERROR=MS002201, * RANGE=(0,'X'FFFF'), * STORE=JAMTVDN </pre>	<pre> 66000001 66050001 66100001 66150001 66200001 66250001 66300001 66350001 66400001 66450001 66500001 66550001 66600001 66650001 66700001 66750001 66800001 66850001 66900001 66950001 </pre>
<pre> 70506 * HCPSTDEF REQMATCH=YES, The VDEV number range start 70507 * MISSING=MS002201 70508 * HCPTKDEF TYPE=HEX, * ERROR=MS002201, * RANGE=(0,'X'FFFF'), * STORE=JAMTVDS 70509 * HCPSTDEF REQMATCH=YES, The VDEV number range end 70510 * MISSING=MS002201 70511 * HCPTKDEF TYPE=HEX, * ERROR=MS002201, * RANGE=(0,'X'FFFF'), * STORE=JAMTVDE </pre>	<pre> 66650001 66700001 66750001 66800001 66850001 66900001 66950001 67000001 67050001 67100001 67150001 67200001 67250001 67300001 </pre>

Table 13. Annotated Listing for Sample CP Exit Routine 2 for CP Exit 1200 (continued)

Assembler Source	Commentary
70513 *-----* 70514 * The HCPDOSYN MACRO dumps the previously defined syntax 70515 *-----*	67400001 67450001 67500001
70517 SYNTAX HCPDOSYN ROOT=YES, BADCMNT=MS670501, BADKWD=MS670502, BADOPN=MS670301, BADQUOT=MS670604, BADREAD=MS670201, BADVERB=MS670102, CONFLICT=MS661301, MAXACC=MS670401, MISSING=MS670401, PLBASE=JAMTABLE, PLLEN=JAMTABLN, PREPROC=0, RETGSDBK=JAMTGSD, SYNLIST=, TOOLONG=MS670702, TOOMANY=MS660304, BASES=PFXPG	X67600001 X67650001 X67700001 X67750001 X67800001 X67850001 X67900001 X67950001 X68000001 X68050001 X68100001 X68150001 X68200001 X68250001 X68300001 X68350001 X68400001 X68450001 68500001
71036 HCPDROPP R0 71038 HCPDROPP R11 71040 HCPDROPP R13 71042 HCPEPILG	68600001 68650001 68700001 68750001

Table 13. Annotated Listing for Sample CP Exit Routine 2 for CP Exit 1200 (continued)

Assembler Source

000078

00071 00A8 71265+JAMSAM

CSECT ,

000078

00071 00A8 71267+HCPDATAA

LOCTR ,

000078

00071 00A8 71268+

LTOrg

000078

01C1D4E3C1C2D3C5

000078

02D6E4D9C3C54040

000078

71269

000088

00061200

=CL(L' ,DRBFIDFN) 'JAMTABLE' ,

000088

000F200

=A(X'F200') ,

000090

00000000

=A(HCPZXUCL) ,

000094

00000000

=A(HCPZXCFC) ,

000098

00000000

=A(HCPZXCLS) ,

00009C

C4C9C140

=CL4'DIA ,

0000A0

0000F200

=A(XIT0F200) ,

0000A4

00000600

=A(HCPZXCAC) ,

0000A8

00000000

=A(HCPZXCLA) ,

0000AC

00000000

=A(HCPZXUDS) ,

0000B0

00000500

=F'1488

0000B4

00000FA0

=FL(L' ,DRBUFESZ) '4000' ,

0000B8

00000050

=F'80

0000BC

00000000

=A(HCPSVCKJ) ,

0000C0

00000000

=A(HCPZIOPN) ,

0000C4

40000000

=A(HCPZIGET) ,

0000C8

40000000

=AL1(C' ,0,0,0) ,

0000CC

00000000

=A(HCPZPRPG) ,

0000D0

00670001

=A(MS670001) ,

0000D4

00670702

=A(MS670702) ,

0000D8

00670201

=A(MS670201) ,

0000DC

00670301

=A(MS670301) ,

0000E0

00000000

=A(HCPZICLS) ,

0000E4

00000FA0

=F'4000' ,

0000E8

00000000

=AL1(0,0,0,0) ,

0000EC

00000000

=A(HCPCVUBM) ,

0000F0

00000000

=A(HCPCVTRN+X'800000000') ,

0000F4

00000000

=A(HCPERMPR) ,

0000F8

00000000

=A(HCPZXUUX) ,

0000FC

00000000

=A(HCPZXUCS) ,

000100

C449

=CL2'D ,

000102

5FFF

=FL(L' ,DRBACSBX) '-1' ,

000104

01FD

=AL(L' ,GSDPRESZ) (FREM) ,

000106

0000

=AL(L' ,DRBRETCD) (DPROK) ,

000108

0058

=AL(L' ,DRBRETCD) (DRBE0F) ,

00010A

01C1D4

=CL3'JAM

00010D

C4C9C1D340

=CL5'DIAL ,

000112

C4C940

=CL3'DI

Line 71267:

Here is another example of generating data out-of-line. LTOrg data is placed early in the module in order to ensure addressability even if the module grows beyond 4 KB in size.

Commentary

Table 14. Annotated Listing of a Local Message Repository for Sample 2 of CP Exit 1200

Assembler Source	Commentary
1 ***** 2 \$ 3 ***** 4 *	 Line 2: CP uses the dollar sign (\$) to indicate that substitution should take place in a message. It is important that you do not allow this indicator to default to any other symbol.
5 670001 E File \$1 \$2 \$3, record \$4 6 * 7 67020101E Error encountered while attempting to read 8 67020101 records from \$1 \$2 \$3 9 * 10 670301 E File \$1 \$2 \$3 not found. 11 *	 Line 5: IBM defines CP message HCP6700E (version 1) as: 670001 E File \$1 \$2, record \$3: This does not provide for the filemode position. We could have: <ul style="list-style-type: none">• Made \$2 = 'ftype fmode'• Ignored showing the filemode• Tried to find another message• Made a mod to HCPMES to add a new message like we want• Made a mod to HCPMES to change 6700.01: 670001 E File \$1 \$2 \$4, record \$3: But none of these would have been in the spirit of CP Exits. What will be generated for this error message is: JANSAM6700E File <i>fn</i> <i>ft</i> <i>fm</i> , record <i>nn</i>

Table 14. Annotated Listing of a Local Message Repository for Sample 2 of CP Exit 1200 (continued)

Assembler Source		Commentary
12 67070201E Statement exceeded maximum length of 4000 on record \$	00001200	Line 12: These repository lines wrap at column 63 just so that the source looks like HCPMES REPOS. To generate a TEXT file from your repository source file, use the CMS command GENMSG. After you have the TEXT file on a CP accessed disk, you can load it with CPXLOAD and associate it with message processing with ASSOCIATE MESSAGE.
13 67070201 4 in file \$1 \$2 \$3.	00001300	
14 *	00001400	
		GENMSG cmpMES REPOS A cmp lang (CP MARGIN 63
15 748007 R Mode ACPBK ACSBK	00001500	Line 15: HPCMPID Our response 7480 version 07. Our command responses for another of our local commands. Message numbers in the 7000-7999 range are considered responses, not error messages, so they never are written with a message header (HCPXXXnnnnE).
16 * R \$1 \$2 \$3	00001600	
17 748008 R \$1 \$2 \$3	00001700	
18 *	00001800	

A Sample JAMTABLE SOURCE File

* Command	Userid	vdev	lo	hi
* -----	-----	----	----	----
?	HELPMACH	*	0000	FFFF
BOOKS	LIBRARY	*	0000	00FF
JOURNALS	LIBRARY	*	0100	01FF
PVM	PVM	*	0000	FFFF
VTAM	MVS	=	0120	013F

All blank records are treated as comments. See line numbers 47517-47522 in the listing.

Records with '*' in column 1 are treated as comments. See line number 47524 in the listing.

Our source file duplicates the entries in the JAM\$DC table that was used in Sample 1, except for the last entry.

The last entry has the value for the *vdev* column equal to '='. This is defined for the parser in the assembler language listing at line number 70503. The parsed value is handled in the assembler language listing at line number 30594.

These are our control data definitions, which are mapped by JAMTABLE. Notice the two entries for commands BOOKS and JOURNALS, where the target user ID is the same but the range of target virtual devices is different. This separation guarantees that neither the BOOKS users nor the JOURNALS users can consume all of the virtual terminal addresses, with one type of use locking out the other.

For this table to be most useful, you should also enter the following DEFINE ALIAS commands:

```

DEFINE ALIAS ?      FOR DIAL ENABLE
DEFINE ALIAS BOOKS  FOR DIAL ENABLE
DEFINE ALIAS JOURNALS FOR DIAL ENABLE
DEFINE ALIAS PVM    FOR DIAL ENABLE
DEFINE ALIAS VTAM   FOR DIAL ENABLE

```

If so, then any of the aliases (?, BOOKS, JOURNALS, PVM, VTAM) or the base command (DIAL) will cause HCPDIA to be called, who will call us as exit 1200.

If a user at a terminal enters the command '?', or 'BOOKS', or any of the other alias names, it will be treated as an alias to DIAL, which means that CP will treat it just as if the user had entered the command 'DIAL' with no operands. Unless exit 1200 supplies a target user ID, this command will fail. It is up to CP Exit 1200 to make the command "whole".

Appendix K. I/O Monitor User Exit

This topic describes how to set up an internal exit routine to monitor real I/O requests.

31-bit address SYSIOMON in HCPSYSCM is provided to point to a user exit routine for the purpose of monitoring real I/O requests started by the CP hypervisor. The intent is for a dynamically loaded user exit module to store and exploit the SYSIOMON address. The SYSIOMON routine address, if nonzero, is called out of the following CP locations:

Module

Call to SYSIOMON

HCPIOS (real I/O dispatcher)

Called just before a real SSCH instruction is executed for any type of real device. Called in two locations.

HCPIQM (real I/O queue manager)

Called just before a real SSCH instruction is executed for any type of real device.

HCPPAU (real paging I/O manager)

Called just before a real SSCH instruction is issued to a real paging or spooling volume.

HCPPAH (real paging interrupt manager)

Called just before a real SSCH instruction is issued to a real paging or spooling volume.

Call Mechanics

If the contents of SYSIOMON are nonzero, representing the address of the user exit routine, the specific HCPCALL invocation to the SYSIOMON routine depends on whether the high order bit, x'80000000', of SYSIOMON is ON or OFF. This was engineered in this manner to be backward compatible over an important change to the HCPCALL invocation to allow the routine to be called from the CP paging or spooling modules, but at the same time not break existing usage of the routine. This operates as follows:

- If the high order bit of SYSIOMON is ON, then the call to the addressed routine is made out of all four CP locations (HCPIOS, HCPIQM, HCPPAU, and HCPPAH). In this case, the call is made as follows, and the addressed user exit routine is expected to conform to the attributes defined on this HCPCALL invocation (that is, static save area, 32-bit registers, 31-bit addressing mode, and run in primary space mode):

```
HCPCALL (R15),ATTR=(STA,SREGE,AM31E,TMSTDE)
```

- If the high order bit of SYSIOMON is OFF, this is considered the legacy call, where the addressed routine is called only out of HCPIOS and HCPIQM. There is no CP paging or spooling call. In this case, the call is made as follows, and the addressed user exit routine is expected to use a dynamic save area (attribute DYN), 32-bit registers (SREGE), 31-bit addressing mode (AM31E), and run in primary space mode (that is, no access register use) (TMSTDE):

```
HCPCALL (R15),TYPE=DIRECT
```

Important Usage Notes

1. An attempt to violate the call attributes as listed in “Call Mechanics” on page 293 will result in an abend or other undesirable, unpredictable results. These attributes were specifically defined to be compatible with the calling modules listed and to conform to their specific requirements, especially serialization requirements.
2. When the exit is called with a static save area, a loss of control in the exit routine is prohibited and will result in an abend or other undesirable, unpredictable results. An example of a loss of control is calling another module which uses a dynamic save area. Be aware that this could occur subtly by a macro, such as with HCPXSERV, and should be avoided.

3. I/O requests to EDEVICES are not monitored by these exit points.
4. I/O done on behalf of the SNAPDUMP command is not monitored by these exit points. SNAPDUMP uses a special purpose I/O dispatcher because the system is quiesced.
5. A SYSIOMON value of zero, the initial value, causes CP to skip the HCPCALL invocation.

Serialization

The calling modules will call the SYSIOMON exit routine holding the following locks:

Module
Locks

HCPIOS

RDEV lock for associated real device is held.

HCPIQM

RDEV lock for associated real device is held.

HCPPAU

RDEV lock for the system volume *might* be held. Exposure block lock (EXPLCKFG.EXPBK) for the volume is held. EXPLCKFG is a spin lock, therefore the exit must not lose control.

HCPPAH

RDEV lock and Exposure block lock (EXPLCKFG.EXPBK) for the system volume are held. EXPLCKFG is a spin lock, therefore the exit must not lose control.

Notices

This information was developed for products and services offered in the US. This material might be available from IBM in other languages. However, you may be required to own a copy of the product or product version in that language in order to access it.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

INTERNATIONAL BUSINESS MACHINES CORPORATION 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 jurisdictions 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. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you provide in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

The performance data and client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information may contain examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

COPYRIGHT LICENSE:

This information may contain sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Programming Interface Information

This book documents intended Programming Interfaces that allow the customer to write programs to obtain the services of z/VM.

Trademarks

IBM, the IBM logo, and [ibm.com](https://www.ibm.com)[®] are trademarks or registered trademarks of International Business Machines Corp., in the United States and/or other countries. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on [IBM Copyright and trademark information](https://www.ibm.com/legal/copytrade) (<https://www.ibm.com/legal/copytrade>).

The registered trademark Linux[®] is used pursuant to a sublicense from the Linux Foundation, the exclusive licensee of Linus Torvalds, owner of the mark on a world-wide basis.

Terms and Conditions for Product Documentation

Permissions for the use of these publications are granted subject to the following terms and conditions.

Applicability

These terms and conditions are in addition to any terms of use for the IBM website.

Personal Use

You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of IBM.

Commercial Use

You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Rights

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

IBM Online Privacy Statement

IBM Software products, including software as a service solutions, ("Software Offerings") may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user, or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering's use of cookies is set forth below.

This Software Offering does not use cookies or other technologies to collect personally identifiable information.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see:

- The section entitled **IBM Websites** at [IBM Privacy Statement](https://www.ibm.com/privacy) (<https://www.ibm.com/privacy>)
- [Cookies and Similar Technologies](https://www.ibm.com/privacy#Cookies_and_Similar_Technologies) (https://www.ibm.com/privacy#Cookies_and_Similar_Technologies)

Bibliography

This topic lists the publications in the z/VM library. For abstracts of the z/VM publications, see [z/VM: General Information](#).

Where to Get z/VM Information

The current z/VM product documentation is available in [IBM Documentation - z/VM \(https://www.ibm.com/docs/en/zvm\)](https://www.ibm.com/docs/en/zvm).

z/VM Base Library

Overview

- [z/VM: License Information](#), GI13-4377
- [z/VM: General Information](#), GC24-6286

Installation, Migration, and Service

- [z/VM: Installation Guide](#), GC24-6292
- [z/VM: Migration Guide](#), GC24-6294
- [z/VM: Service Guide](#), GC24-6325
- [z/VM: VMSES/E Introduction and Reference](#), GC24-6336

Planning and Administration

- [z/VM: CMS File Pool Planning, Administration, and Operation](#), SC24-6261
- [z/VM: CMS Planning and Administration](#), SC24-6264
- [z/VM: Connectivity](#), SC24-6267
- [z/VM: CP Planning and Administration](#), SC24-6271
- [z/VM: Getting Started with Linux on IBM Z](#), SC24-6287
- [z/VM: Group Control System](#), SC24-6289
- [z/VM: I/O Configuration](#), SC24-6291
- [z/VM: Running Guest Operating Systems](#), SC24-6321
- [z/VM: Saved Segments Planning and Administration](#), SC24-6322
- [z/VM: Secure Configuration Guide](#), SC24-6323

Customization and Tuning

- [z/VM: CP Exit Customization](#), SC24-6269
- [z/VM: Performance](#), SC24-6301

Operation and Use

- [z/VM: CMS Commands and Utilities Reference](#), SC24-6260
- [z/VM: CMS Primer](#), SC24-6265
- [z/VM: CMS User's Guide](#), SC24-6266
- [z/VM: CP Commands and Utilities Reference](#), SC24-6268

- [z/VM: System Operation](#), SC24-6326
- [z/VM: Virtual Machine Operation](#), SC24-6334
- [z/VM: XEDIT Commands and Macros Reference](#), SC24-6337
- [z/VM: XEDIT User's Guide](#), SC24-6338

Application Programming

- [z/VM: CMS Application Development Guide](#), SC24-6256
- [z/VM: CMS Application Development Guide for Assembler](#), SC24-6257
- [z/VM: CMS Application Multitasking](#), SC24-6258
- [z/VM: CMS Callable Services Reference](#), SC24-6259
- [z/VM: CMS Macros and Functions Reference](#), SC24-6262
- [z/VM: CMS Pipelines User's Guide and Reference](#), SC24-6252
- [z/VM: CP Programming Services](#), SC24-6272
- [z/VM: CPI Communications User's Guide](#), SC24-6273
- [z/VM: ESA/XC Principles of Operation](#), SC24-6285
- [z/VM: Language Environment User's Guide](#), SC24-6293
- [z/VM: OpenExtensions Advanced Application Programming Tools](#), SC24-6295
- [z/VM: OpenExtensions Callable Services Reference](#), SC24-6296
- [z/VM: OpenExtensions Commands Reference](#), SC24-6297
- [z/VM: OpenExtensions POSIX Conformance Document](#), GC24-6298
- [z/VM: OpenExtensions User's Guide](#), SC24-6299
- [z/VM: Program Management Binder for CMS](#), SC24-6304
- [z/VM: Reusable Server Kernel Programmer's Guide and Reference](#), SC24-6313
- [z/VM: REXX/VM Reference](#), SC24-6314
- [z/VM: REXX/VM User's Guide](#), SC24-6315
- [z/VM: Systems Management Application Programming](#), SC24-6327
- [z/VM: z/Architecture Extended Configuration \(z/XC\) Principles of Operation](#), SC27-4940

Diagnosis

- [z/VM: CMS and REXX/VM Messages and Codes](#), GC24-6255
- [z/VM: CP Messages and Codes](#), GC24-6270
- [z/VM: Diagnosis Guide](#), GC24-6280
- [z/VM: Dump Viewing Facility](#), GC24-6284
- [z/VM: Other Components Messages and Codes](#), GC24-6300
- [z/VM: VM Dump Tool](#), GC24-6335

z/VM Facilities and Features

Data Facility Storage Management Subsystem for z/VM

- [z/VM: DFSMS/VM Customization](#), SC24-6274
- [z/VM: DFSMS/VM Diagnosis Guide](#), GC24-6275
- [z/VM: DFSMS/VM Messages and Codes](#), GC24-6276
- [z/VM: DFSMS/VM Planning Guide](#), SC24-6277

- *z/VM: DFSMS/VM Removable Media Services*, SC24-6278
- *z/VM: DFSMS/VM Storage Administration*, SC24-6279

Directory Maintenance Facility for z/VM

- *z/VM: Directory Maintenance Facility Commands Reference*, SC24-6281
- *z/VM: Directory Maintenance Facility Messages*, GC24-6282
- *z/VM: Directory Maintenance Facility Tailoring and Administration Guide*, SC24-6283

Open Systems Adapter

- Open Systems Adapter/Support Facility on the Hardware Management Console (https://www.ibm.com/docs/en/SSLTBW_2.3.0/pdf/SC14-7580-02.pdf), SC14-7580
- Open Systems Adapter-Express ICC 3215 Support (<https://www.ibm.com/docs/en/zos/2.3.0?topic=osa-icc-3215-support>), SA23-2247
- Open Systems Adapter Integrated Console Controller User's Guide (https://www.ibm.com/docs/en/SSLTBW_2.3.0/pdf/SC27-9003-02.pdf), SC27-9003
- Open Systems Adapter-Express Customer's Guide and Reference (https://www.ibm.com/docs/en/SSLTBW_2.3.0/pdf/iaa2z1f0.pdf), SA22-7935

Performance Toolkit for z/VM

- *z/VM: Performance Toolkit Guide*, SC24-6302
- *z/VM: Performance Toolkit Reference*, SC24-6303

The following publications contain sections that provide information about z/VM Performance Data Pump, which is licensed with Performance Toolkit for z/VM.

- *z/VM: Performance*, SC24-6301. See *z/VM Performance Data Pump*.
- *z/VM: Other Components Messages and Codes*, GC24-6300. See *Data Pump Messages*.

RACF® Security Server for z/VM

- *z/VM: RACF Security Server Auditor's Guide*, SC24-6305
- *z/VM: RACF Security Server Command Language Reference*, SC24-6306
- *z/VM: RACF Security Server Diagnosis Guide*, GC24-6307
- *z/VM: RACF Security Server General User's Guide*, SC24-6308
- *z/VM: RACF Security Server Macros and Interfaces*, SC24-6309
- *z/VM: RACF Security Server Messages and Codes*, GC24-6310
- *z/VM: RACF Security Server Security Administrator's Guide*, SC24-6311
- *z/VM: RACF Security Server System Programmer's Guide*, SC24-6312
- *z/VM: Security Server RACROUTE Macro Reference*, SC24-6324

Remote Spooling Communications Subsystem Networking for z/VM

- *z/VM: RSCS Networking Diagnosis*, GC24-6316
- *z/VM: RSCS Networking Exit Customization*, SC24-6317
- *z/VM: RSCS Networking Messages and Codes*, GC24-6318
- *z/VM: RSCS Networking Operation and Use*, SC24-6319
- *z/VM: RSCS Networking Planning and Configuration*, SC24-6320

TCP/IP for z/VM

- [*z/VM: TCP/IP Diagnosis Guide*](#), GC24-6328
- [*z/VM: TCP/IP LDAP Administration Guide*](#), SC24-6329
- [*z/VM: TCP/IP Messages and Codes*](#), GC24-6330
- [*z/VM: TCP/IP Planning and Customization*](#), SC24-6331
- [*z/VM: TCP/IP Programmer's Reference*](#), SC24-6332
- [*z/VM: TCP/IP User's Guide*](#), SC24-6333

Prerequisite Products

Device Support Facilities

- Device Support Facilities (ICKDSF): User's Guide and Reference (https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/ickug00_v2r5.pdf), GC35-0033

Related Products

XL C++ for z/VM

- [*XL C/C++ for z/VM: Runtime Library Reference*](#), SC09-7624
- [*XL C/C++ for z/VM: User's Guide*](#), SC09-7625

z/OS

IBM Documentation - z/OS (<https://www.ibm.com/docs/en/zos>)

Index

Numerics

3277 display
exit for installation-supplied code [166](#)

A

accounting
exit, CP [165](#)
acquire
lock on component ID [170](#)
output data from CP exit routine [12](#)
add
configuration file statements [77](#)
new
alias [49](#)
CP commands [4](#), [43](#)
CP exit routines [4](#)
CP message repositories [4](#), [51](#)
diagnose codes [4](#), [43](#)
address constant resolution
definition [31](#)
effect on
CPXLOAD of prohibiting [29](#)
CPXLOAD when unresolved [31](#)
SYSGEN when unresolved [31](#)
when resolved by CPXLOAD [29](#)
addresses, storage [2](#)
addressing mode
addressing mode [14](#)
choosing for CP exit routine [13](#)
entry point attributes [14](#)
register styles [13](#)
translation mode [14](#)
alias
creating [49](#)
allocate
CMPBKs [170](#)
component ID blocks [170](#)
ALLOCATE operand
HCPXSERV macro [172](#)
alter
CP load list [213](#)
CP mods to CP exits [9](#)
LOGON command operands [109](#)
return code from parsing LOGON command [112](#)
separator page [145](#)
system initialization options [101](#)
VMDBK
after creation/initialization [114](#)
alternate MDLAT
coding [213](#)
creating [214](#)
using [16](#)
AMODE31 operand
MDLATENT macro [182](#)
AMODE31ENT operand

AMODE31ENT operand (*continued*)
MDLATENT macro [184](#)
AMODE64 operand
MDLATENT macro [182](#)
AMODE64ENT operand
MDLATENT macro [184](#)
AMODEINT operand
MDLATENT macro [183](#)
AMODEVAR operand
MDLATENT macro [182](#)
AMODEVARENT operand
MDLATENT macro [184](#)
AOFARM parameter list [167](#)
AONPARM parameter list [166](#)
ASSOCIATE EXIT command [41](#)
ASSOCIATE EXIT statement [41](#)
ASSOCIATE MESSAGES command [41](#), [54](#)
ASSOCIATE MESSAGES statement [41](#)
ASSOCIATE MSGS command [41](#), [54](#)
ASSOCIATE MSGS statement [41](#)
attribute list
defining
for CP modules [181](#)
for entry points [181](#)
marking
the beginning of [186](#)
the end of [187](#)
restrictions for CP routines [185](#)
attributes
addressing mode [14](#)
entry point [14](#)
register style [13](#)
translation mode [14](#)
authorization
examining CP commands after [105](#)

B

beginning
CP module attribute list, marking [186](#)
block, command table entry
changing after authorization processing [105](#)
creating with COMMD macro [7](#)

C

call
CP exit points [4](#)
CP exit routines [170](#)
local CP message repositories [4](#), [25](#)
CALL operand
HCPXSERV macro [171](#)
change
CP load list [213](#)
CP mods to CP exits [9](#)
LOGON command operands [109](#)
return code from parsing LOGON command [112](#)

- change (*continued*)
 - separator page [145](#)
 - system initialization options [101](#)
 - VMDBK
 - after creation/initialization [114](#)
- CHANGE directive [152](#)
- choose
 - addressing mode for CP exit routine [13](#)
- class
 - IBM [44](#)
 - privilege [44](#)
- code, diagnose
 - controlling [4](#), [40](#)
 - creating [4](#), [46](#)
 - overriding [4](#)
- code, executable
 - dynamically loading into system execution space [3](#), [33](#)
 - dynamically unloading from system execution space [3](#), [61](#)
- code, return
 - changing
 - from parsed LOGON command [112](#)
 - examining
 - from parsed LOGON command [112](#)
 - standard [94](#)
- command
 - ASSOCIATE
 - EXIT [41](#)
 - MESSAGES [41](#)
 - MSGS [41](#)
 - controlling [4](#), [39](#)
 - CPLISTFILE [86](#), [87](#), [156](#)
 - CPTYPE [87](#)
 - CPXLOAD
 - examples [35](#)
 - loading CP command routines [39](#)
 - loading CP exit routines [40](#)
 - loading diagnose code routines [40](#)
 - loading message repositories [41](#)
 - operand explanations [33](#)
 - restrictions for CP routine attributes [185](#)
 - CPXUNLOAD
 - unloading CP command routines [39](#)
 - unloading CP exit routines [40](#)
 - unloading diagnose code routines [40](#)
 - unloading message repositories [41](#)
 - creating [4](#), [43](#)
 - DEFINE
 - ALIAS [39](#)
 - CMD [39](#)
 - COMMAND [39](#)
 - DIAGNOSE [40](#)
 - EXIT [40](#)
 - DISABLE
 - CMD [39](#)
 - COMMAND [39](#)
 - DIAGNOSE [40](#)
 - EXITS [40](#)
 - ENABLE
 - CMD [39](#)
 - COMMAND [39](#)
 - DIAGNOSE [40](#)
 - EXITS [40](#)
 - LOCATE

- command (*continued*)
 - LOCATE (*continued*)
 - SYMBOL [39–41](#)
 - LOGOFF
 - examining operands [117](#), [119](#)
 - performing processing before completion [119](#)
 - LOGON
 - changing operands [109](#)
 - changing return code after parsing [112](#)
 - examining operands [109](#), [116](#)
 - examining return code after parsing [112](#)
 - performing processing before completion [116](#)
 - rejecting [109](#)
 - replacing operands [109](#)
 - MODIFY
 - CMD [39](#)
 - COMMAND [39](#)
 - DIAGNOSE [40](#)
 - EXIT [41](#)
 - overriding [4](#)
 - QUERY
 - CPCMDs [39](#), [49](#)
 - DIAGNOSE [40](#)
- command table entry block
 - changing after authorization processing [105](#)
 - creating with COMMD macro [7](#)
- COMPID operand
 - HCPXSERV macro [173](#)
- component ID block
 - acquiring lock on [170](#)
 - allocating [170](#)
 - controlling access to [170](#)
 - deallocating [170](#)
 - locating [170](#)
- configuration file, system
 - ASSOCIATE
 - EXIT [41](#)
 - MESSAGES [41](#)
 - MSGS [41](#)
 - CPXLOAD statement
 - loading CP command routines [39](#)
 - loading CP exit routines [40](#)
 - loading diagnose code routines [40](#)
 - loading message repositories [41](#)
 - parameter explanations [33](#)
 - restrictions for CP routine attributes [185](#)
 - DEFINE statement
 - ALIAS [39](#)
 - CMD [39](#)
 - COMMAND [39](#)
 - DIAGNOSE [40](#)
 - EXIT [40](#)
 - DISABLE statement
 - CMD [39](#)
 - COMMAND [39](#)
 - DIAGNOSE [40](#)
 - EXITS [40](#)
 - ENABLE statement
 - CMD [39](#)
 - COMMAND [39](#)
 - DIAGNOSE [40](#)
 - EXITS [40](#)
 - MODIFY statement
 - CMD [39](#)

configuration file, system (*continued*)

MODIFY statement (*continued*)

COMMAND [39](#), [40](#)

EXIT [41](#)

console input data

replacing [109](#)

constant resolution, address

definition [31](#)

effect on

CPXLOAD of prohibiting [29](#)

CPXLOAD when unresolved [31](#)

SYSGEN when unresolved [31](#)

when resolved by CPXLOAD [29](#)

control

access to component ID block [170](#)

CP commands [4](#), [39](#)

CP exit points [4](#), [40](#)

diagnose codes [4](#), [40](#)

dynamically loaded routine [39](#)

local CP message repositories [4](#), [41](#)

control block

CMPBK

allocating [170](#)

controlling access to [170](#)

deallocating [170](#)

locating [170](#)

relinquishing lock on [170](#)

XITBK [170](#)

CONTROL operand

OPTIONS directive [158](#)

when to specify [34](#)

control section (CSECT)

size, increasing [154](#)

convert

CP load list [213](#)

CP mods to CP exits [9](#)

LOGON command operands [109](#)

return code from parsing LOGON command [112](#)

separator page [145](#)

system initialization options [101](#)

VMDBK

after creation/initialization [114](#)

CP (control program)

commands

examining after authorization [105](#)

customization

CP Exit method [3](#)

SYSGEN method [2](#)

exit points

calling [4](#)

controlling [4](#), [40](#)

CP Exit 00E0 [97](#)

CP Exit 00E1 [101](#)

CP Exit 0FFB [105](#)

CP Exit 1100 [109](#)

CP Exit 1101 [112](#)

CP Exit 1110 [114](#)

CP Exit 117F [116](#)

CP Exit 11C0 [117](#)

CP Exit 11FF [119](#)

CP Exit 1200 [120](#)

CP Exit 1201 [122](#)

CP Exit 1210 [123](#), [129](#), [130](#)

CP Exit 3FE8 [131](#)

CP (control program) (*continued*)

exit points (*continued*)

CP Exit 4400 [133](#)

CP Exit 4401 [135](#)

CP Exit 4402 [137](#)

CP Exit 4403 [138](#)

CP Exit 4404 [140](#)

CP Exit 4405 [142](#)

CP Exit 4406 [143](#)

CP Exit 4407 [145](#)

defining parameter list [170](#)

defining using HCPXSERV macro [4](#)

dynamic [6](#), [149](#)

exit routines

choosing addressing mode for [13](#)

creating [4](#)

providing input data to [12](#)

receiving output data from [12](#)

macros

HCPXSERV [170](#)

MDLATENT [181](#)

MDLATHDR [186](#)

MDLATTLR [187](#)

CP (Control Program)

accounting exit [165](#)

exits

accounting records [165](#)

HCPACU module [165](#)

macros

HCPDROP [166](#)

HCPENTER [166](#)

HCPEPILG [166](#)

HCPEXIT [166](#)

HCPPROLG [166](#)

HCPUSING [166](#)

CP customization

CP Exit method [3](#)

SYSGEN method [2](#)

CP Exit 00E0 (Startup Pre-Prompt Processing) [97](#)

CP Exit 00E1 (Startup Post-Prompt Processing) [101](#)

CP Exit 0FFB (Post-Authorization Command Processing) [105](#)

CP Exit 1100 (LOGON Command Pre-Parse Processing) [109](#)

CP Exit 1101 (Logon Post-Parse Processing) [112](#)

CP Exit 1110 (VMDBK Pre-Logon Processing) [114](#)

CP Exit 117F (Logon Final Screening) [116](#)

CP Exit 11C0 (Logoff Initial Screening) [117](#)

CP Exit 11FF (Logoff Final Screening) [119](#)

CP Exit 1200 (DIAL Command Initial Screening) [120](#)

CP Exit 1201 (DIAL Command Final Screening) [122](#)

CP Exit 1210 (Messaging Commands Screening) [123](#), [129](#), [130](#)

CP Exit 3FE8 (SHUTDOWN Command Screening) [131](#)

CP Exit 4400 (Separator Page Data Customization) [133](#)

CP Exit 4401 (Separator Page Pre-Perforation Positioning) [135](#)

CP Exit 4402 (Separator Page Perforation Printing or 3800 Positioning) [137](#)

CP Exit 4403 (Separator Page Printing) [138](#)

CP Exit 4404 (Second Separator Page Positioning) [140](#)

CP Exit 4405 (Second Separator Page Printing) [142](#)

CP Exit 4406 (Separator Page Post-Print Positioning) [143](#)

CP Exit 4407 (Trailer Page Processing) [145](#)

CP exit point

benefits of using [1](#)

- CP exit point (*continued*)
 - calling [4](#)
 - controlling [4](#), [40](#)
 - defining
 - parameter list [170](#)
 - using HCPXSERV macro [4](#), [179](#)
 - disabling [170](#)
 - dynamic [6](#), [149](#)
 - enabling [170](#)
 - locating control blocks [170](#)
 - testing [170](#)
- CP exit routine
 - choosing addressing mode for [13](#)
 - creating [4](#)
 - providing input data to [12](#)
 - receiving output data from [12](#)
- CP load list
 - updating [213](#)
- CP message repository
 - advantages of [51](#)
 - calling local [4](#), [25](#)
 - commenting [209](#)
 - control line [209](#)
 - controlling local [4](#), [41](#)
 - creating local [4](#), [51](#)
 - example [52](#)
 - message record format [209](#)
 - overview [209](#)
 - using local [4](#)
- CP modifications
 - migrating to CP Exits [5](#)
- CP module
 - attribute list
 - defining [181](#)
 - marking the beginning of [186](#)
 - marking the end of [187](#)
 - restrictions for CP routines [185](#)
 - attributes
 - addressing mode [14](#)
 - register style [13](#)
 - translation mode [14](#)
 - defining attribute list [181](#)
 - packaging [35](#)
 - writing [35](#)
- CP nucleus
 - ordered module list [213](#)
- CP parser
 - creating a syntax definition [64](#)
 - HCPCFDEF macro [63](#), [190](#)
 - HCPDOSYN macro [63](#), [191](#)
 - HCPSTDEF macro [63](#), [193](#)
 - HCPTKDEF macro [63](#), [196](#)
 - overview [63](#), [189](#)
 - syntax descriptions [63](#)
 - token conversion types [63](#), [198](#)
- CP service
 - available for CP Exit 00E0 [100](#)
 - available for CP Exit 00E1 [104](#)
- CPLISTFILE command [86](#), [87](#), [156](#)
- CPTYPE command [87](#)
- CPXLOAD command
 - examples [35](#)
 - loading
 - CP command routines [39](#)

- CPXLOAD command (*continued*)
 - loading (*continued*)
 - CP exit routines [40](#)
 - diagnose code routines [40](#)
 - message repositories [41](#)
 - message repositories [54](#)
 - operand explanations [33](#)
 - restrictions for CP routine attributes [185](#)
- CPXLOAD directive
 - CHANGE [152](#)
 - examples [37](#)
 - EXPAND [154](#)
 - INCLUDE [156](#)
 - OPTIONS [158](#)
- CPXLOAD operand
 - MDLATENT macro [185](#)
- CPXLOAD statement
 - loading
 - CP command routines [39](#)
 - CP exit routines [40](#)
 - diagnose code routines [40](#)
 - message repositories [41](#)
 - loading CP exit routines [40](#)
 - loading diagnose code routines [40](#)
 - loading message repositories [41](#)
 - parameter explanations [33](#)
 - restrictions for CP routine attributes [185](#)
- CPXUNLOAD command
 - unloading
 - CP command routines [39](#)
 - CP exit routines [40](#)
 - diagnose code routines [40](#)
 - message repositories [41](#)
- create
 - dynamically loaded routines [11](#)
 - new
 - alias [49](#)
 - CP commands [4](#)
 - CP exit routines [4](#)
 - CP message repositories [4](#)
 - diagnose codes [4](#)
 - parameter list for CP exit point [170](#)
 - VMDBK
 - changing after [114](#)
 - examining after [114](#)
- CSECT
 - size, increasing [154](#)
- customization, CP
 - CP Exit method [3](#)
 - SYSGEN method [2](#)

D

- DATA operand
 - MDLATENT macro [182](#)
- data, console input
 - replacing [109](#)
- date, input
 - providing to CP exit routine [12](#)
- date, output
 - receiving from CP exit routine [12](#)
- deallocate
 - CMPBKs [170](#)
 - component ID blocks [170](#)

- DEALLOCATE operand
 - HCPXSERV macro [173](#)
- debug
 - CP Exit 00E0 [101](#)
 - CP Exit 00E1 [105](#)
- default
 - setting for loading CP routines [158](#)
- define
 - CP module
 - attribute list [181](#)
 - defaults for loading CP routines [158](#)
 - lock on component ID [170](#)
 - new
 - alias [49](#)
 - CP commands [4](#)
 - CP exit routines [4](#)
 - CP message repositories [4](#)
 - diagnose codes [4](#)
 - parameter list for CP exit point [170](#)
- DEFINE ALIAS command [39](#)
- DEFINE ALIAS statement [39](#)
- DEFINE CMD command [39](#)
- DEFINE CMD statement [39](#)
- DEFINE COMMAND command [39](#)
- DEFINE COMMAND statement [39](#)
- DEFINE DIAGNOSE command [40](#)
- DEFINE DIAGNOSE statement [40](#)
- DEFINE EXIT command [40](#)
- DEFINE EXIT statement [40](#)
- definition of
 - address constant resolution [31](#)
 - dynamically loaded CP routines [1](#)
 - SXS dynamic area [2](#)
- DELAY operand
 - when to specify [33](#)
- delete
 - external symbol, temporarily [152](#)
- diagnose code
 - controlling [4](#), [40](#)
 - creating [4](#), [46](#)
 - overriding [4](#)
- directive, CPXLOAD
 - CHANGE [152](#)
 - examples [37](#)
 - EXPAND [154](#)
 - INCLUDE [156](#)
 - OPTIONS [158](#)
- disable
 - CP exit points [170](#)
- DISABLE CMD command [39](#)
- DISABLE CMD statement [39](#)
- DISABLE COMMAND command [39](#)
- DISABLE COMMAND statement [39](#)
- DISABLE DIAGNOSE command [40](#)
- DISABLE DIAGNOSE statement [40](#)
- DISABLE EXITS command [40](#)
- DISABLE EXITS statement [40](#)
- DISABLE operand
 - HCPXSERV macro [171](#)
- DISABLED operand
 - HCPXSERV macro [171](#)
- DISASSOCIATE EXIT command [41](#)
- DISASSOCIATE MESSAGES command [41](#), [61](#)
- display

- display (*continued*)
 - messages with HCPCONSL macro [9](#)
- dynamic
 - loading
 - deleting external symbol, temporarily [152](#)
 - including another file [156](#)
 - into SXS dynamic area [3](#)
 - renaming external symbol, temporarily [152](#)
 - restrictions for CP routine attributes [185](#)
 - routines into SXS dynamic area [33](#)
 - setting defaults for loading [158](#)
 - unloading
 - from SXS dynamic area [3](#)
- dynamic area, SXS
 - definition [2](#)
 - loading code into [3](#), [33](#)
 - unloading code from [3](#)
- dynamic exit point
 - conventions [149](#)
 - defining [6](#), [40](#)
 - modifying [41](#)
 - querying [41](#)
 - removing [41](#)
- DYNAMIC operand
 - MDLATENT macro [183](#)
- dynamically-loaded routine
 - choosing addressing mode for [13](#)
 - controlling [39](#)
 - creating [11](#)
 - definition [1](#)
 - deleting external symbol for, temporarily [152](#)
 - design issues [11](#)
 - including another file when loading [156](#)
 - providing input data to [12](#)
 - receiving output data from [12](#)
 - renaming external symbol for, temporarily [152](#)
 - restrictions for CP routine attributes [185](#)
 - setting defaults for loading [158](#)

E

- enable
 - CP exit points [170](#)
- ENABLE CMD command [39](#)
- ENABLE CMD statement [39](#)
- ENABLE COMMAND command [39](#)
- ENABLE COMMAND statement [39](#)
- ENABLE DIAGNOSE command [40](#)
- ENABLE DIAGNOSE statement [40](#)
- ENABLE EXITS command [40](#)
- ENABLE EXITS statement [40](#)
- ENABLE operand
 - HCPXSERV macro [171](#)
- end
 - CP module attribute list, marking [187](#)
- entry block, command table
 - changing after authorization processing [105](#)
 - creating with COMMD macro [7](#)
- entry point
 - attribute list
 - defining [181](#)
 - restrictions for CP routines [185](#)
 - attributes [14](#)
- EP operand

- EP operand (*continued*)
 - MDLATENT macro [183](#)
- ERROR operand
 - HCPXSERV macro [171](#)
- examine
 - CP commands after authorization [105](#)
 - LOGOFF command [117](#), [119](#)
 - LOGON command operands [116](#)
 - LOGON COMMAND operands [109](#)
 - return code from parsing LOGON command [112](#)
 - system initialization options [101](#)
 - VMDBK
 - after creation/initialization [114](#)
- example
 - acquiring
 - exclusive lock on CMPBK [179](#)
 - shared lock on CMPBK [178](#)
 - allocating
 - CMPBK [179](#)
 - component ID block [179](#)
 - parameter list build area [179](#)
 - PLISTBLD [179](#)
 - calling
 - CP exit routines [179](#)
 - changing
 - external symbols when loading CP routines [152](#)
 - configuration file statement [77](#)
 - CP message repository [52](#)
 - CP routines, setting defaults for loading [159](#)
 - CPXLOAD commands [35](#)
 - CPXLOAD directives [37](#)
 - CSECT size, expanding [154](#)
 - defining
 - more CMPBK space [179](#)
 - deleting
 - external symbols when loading CP routines [152](#)
 - expanding
 - CSECT size [154](#)
 - finding
 - CMPBK [178](#)
 - component ID block [178](#)
 - including
 - another file during loading [157](#)
 - increasing
 - CSECT size [154](#)
 - loading
 - CP routines, setting defaults [159](#)
 - locating
 - CMPBK [178](#)
 - component ID block [178](#)
 - passing
 - control blocks [179](#)
 - parameter list entries [179](#)
 - PLIST entries [179](#)
 - releasing
 - parameter list build area [179](#)
 - PLISTBLD [179](#)
 - renaming
 - external symbols when loading CP routines [152](#)
 - setting
 - defaults for loading CP routines [159](#)
 - size, expanding CSECT [154](#)
- executable code
 - dynamically loading into system execution space [3](#), [33](#)

- executable code (*continued*)
 - dynamically unloading from system execution space [3](#), [61](#)
- execution space, system
 - dynamic area
 - definition [2](#)
 - loading code into [3](#), [33](#)
 - unloading code from [3](#)
 - layout [2](#)
- EXIT operand
 - HCPXSERV macro [171](#)
- exit point, CP
 - benefits of using [1](#)
 - calling [4](#)
 - controlling [4](#), [40](#)
 - defining
 - parameter list [170](#)
 - using HCPXSERV macro [4](#), [179](#)
 - disabling [170](#)
 - dynamic [6](#), [149](#)
 - enabling [170](#)
 - locating control blocks [170](#)
 - testing [170](#)
- exit routine, CP
 - choosing addressing mode for [13](#)
 - creating [4](#)
 - providing input data to [12](#)
 - receiving output data from [12](#)
- exits
 - accounting
 - CP [165](#)
 - CP
 - accounting records [165](#)
 - HCPACU module [165](#)
 - HCPACU module
 - contents [165](#)
 - entry point HCPACUOF [167](#)
 - entry point HCPACUON [166](#)
 - function [165](#)
 - usage notes [165](#)
 - when called by CP [165](#)
 - installation-wide
 - accounting, CP [165](#)
 - CP accounting records [165](#)
 - HCPACU module [165](#)
- EXPAND directive [154](#)
- external symbol
 - deleting temporarily [152](#)
 - renaming temporarily [152](#)
- EXTERNAL_SYNTAX statement [77](#)

F

- FASTDYN operand
 - MDLATENT macro [183](#), [184](#)
- file
 - CP message repository
 - advantages of [51](#)
 - commenting [209](#)
 - control line [209](#)
 - example [52](#)
 - message record format [209](#)
 - overview [209](#)
 - including another when loading code [156](#)

file, system configuration

ASSOCIATE

EXIT [41](#)

MESSAGES [41](#)

MSGs [41](#)

CPXLOAD statement

loading CP command routines [39](#)

loading CP exit routines [40](#)

loading diagnose code routines [40](#)

loading message repositories [41](#)

parameter explanations [33](#)

restrictions for CP routine attributes [185](#)

DEFINE statement

ALIAS [39](#)

CMD [39](#)

COMMAND [39](#)

DIAGNOSE [40](#)

EXIT [40](#)

DISABLE statement

CMD [39](#)

COMMAND [39](#)

DIAGNOSE [40](#)

EXITS [40](#)

ENABLE statement

CMD [39](#)

COMMAND [39](#)

DIAGNOSE [40](#)

EXITS [40](#)

MODIFY statement

CMD [39](#)

COMMAND [39](#), [40](#)

EXIT [41](#)

find

CMPBKs [170](#)

component ID blocks [170](#)

CP exit control blocks [170](#)

XITBKs [170](#)

FIND operand

HCPXSERV macro [173](#)

format

record

messages [209](#)

G

get

lock on component ID [170](#)

output data from CP exit routine [12](#)

give

input data to CP exit routine [12](#)

system initialization options [97](#)

GOTO operand

MDLATENT macro [184](#)

H

HCP002E [159](#)

HCP2757E [159](#)

HCP2768E [159](#)

HCP6704E [160](#)

HCP8019E [160](#)

HCP8040E [160](#)

HCPACU exit module

HCPACU exit module (*continued*)

contents [165](#)

function [165](#)

HCPACUOF entry point

AOFARM parameter list [167](#)

function [167](#)

input registers [166](#), [167](#)

return code [168](#)

when called by CP [167](#)

HCPACUON entry point

AONPARM parameter list [166](#)

function [166](#)

input registers [166](#)

return codes [167](#)

when called by CP [166](#)

usage notes [165](#)

when called by CP [165](#)

HCPAIF, HCPAELSE, HCPAEND [83](#)

HCPCFDEF macro [63](#)

HCPCMPID macro

using [16](#)

HCPCONSL macro

displaying messages with [9](#)

HCPDOSYN macro [191](#)

HCPDROP macro

HCPACU module [166](#)

HCPENTER macro

HCPACU module [166](#)

HCPEPILG macro

HCPACU module [166](#)

HCPEXIT macro

HCPACU module [166](#)

HCPPROLG macro

HCPACU module [166](#)

HCPSAVBK save area

contents [166](#)

used by CP accounting exit [166](#)

HCPSTDEF macro [193](#)

HCPTKDEF macro [196](#)

HCPTRANS macro [9](#)

HCPUSING macro, HCPACU module [166](#)

HCPXSERV macro

defining CP exit points with [4](#)

general definition [170](#)

Host Absolute Address (HAA) [2](#)

Host Logical Address (HLA) [2](#)

Host Real Address (HRA) [2](#)

I

I/O monitor user exit [293](#)

identify

input data to CP exit routine [12](#)

system initialization options [97](#)

include

another file when loading code [156](#)

loop, preventing [156](#)

INCLUDE directive [156](#)

increase

CSECT size [154](#)

initialization options, system

changing [101](#)

examining [101](#)

providing [97](#)

initialization options, system (*continued*)

when prompted for [99](#), [103](#)

initialize

VMDBK

changing after [114](#)

examining after [114](#)

input data

providing to CP exit routine [12](#)

input data, console

replacing [109](#)

INVEXIT operand

HCPXSERV macro [171](#)

IPL

dynamically loading your CP routines during [33](#)

system initialization options

changing [101](#)

examining [101](#)

providing [97](#)

when prompted for [99](#), [103](#)

L

language

translating your messages into another [51](#)

LET operand

OPTIONS directive [159](#)

when to specify [34](#)

list

parameter, defining for CP exit point [170](#)

LLATTR operand

MDLATENT macro [185](#)

load

dynamically

deleting external symbol, temporarily [152](#)

including another file [156](#)

into SXS [3](#)

renaming external symbol, temporarily [152](#)

restrictions for CP routine attributes [185](#)

routines into SXS dynamic area [33](#)

setting defaults for loading [158](#)

loaded routine, dynamically

choosing addressing mode for [13](#)

controlling [39](#)

creating [11](#)

definition [1](#)

deleting external symbol for, temporarily [152](#)

design issues [11](#)

including another file when loading [156](#)

providing input data to [12](#)

receiving output data from [12](#)

renaming external symbol for, temporarily [152](#)

restrictions for CP routine attributes [185](#)

setting defaults for loading [158](#)

local message repository

calling CP [4](#)

controlling CP [4](#), [41](#)

creating CP [4](#)

using CP [4](#)

locate

CMPBKs [170](#)

component ID blocks [170](#)

CP exit control blocks [170](#)

XITBKs [170](#)

LOCATE CMDBK command [81](#)

LOCATE DGNBK command [81](#)

LOCATE ICLBK command [81](#)

LOCATE SYMBOL command [39–41](#)

LOCATE XITBK command [81](#)

lock

acquiring on component ID [170](#)

relinquishing on component ID [170](#)

LOCK operand

OPTIONS directive [159](#)

LOCKEXCLUSIVE operand

HCPXSERV macro [173](#)

LOCKSHARED operand

HCPXSERV macro [173](#)

log off

examining characteristics of [117](#), [119](#)

performing processing before completion [119](#)

log on

changing

characteristics of [109](#)

return code after parsing [112](#)

VMDBK during [114](#)

examining

characteristics of [109](#), [116](#)

return code after parsing [112](#)

VMDBK during [114](#)

performing processing before completion [116](#)

rejecting [109](#)

LOGOFF command

examining operands [117](#), [119](#)

performing processing before completion [119](#)

LOGON command

changing

operands [109](#)

return code after parsing [112](#)

examining

operands [109](#), [116](#)

return code after parsing [112](#)

performing processing before completion [116](#)

rejecting [109](#)

replacing operands [109](#)

LONGREG operand

MDLATENT macro [182](#)

LONGREGENT operand

MDLATENT macro [184](#)

loop

include

preventing [156](#)

system initialization

preventing for CP Exit 00E0 [100](#)

preventing for CP Exit 00E1 [104](#)

M

macroinstruction

CP

HCPDROP [166](#)

HCPENTER [166](#)

HCPEPILG [166](#)

HCPEXIT [166](#)

HCPPROLG [166](#)

HCPUSING [166](#)

HCPXSERV [170](#)

MDLATENT [181](#)

MDLATHDR [186](#)

- macroinstruction (*continued*)
 - CP (*continued*)
 - MDLATTR 187
- mark
 - end of CP module attribute list 187
 - start of CP module attribute list 186
- MDLATENT macro 181
- MDLATHDR macro 186
- MDLATTR macro 187
- MEMBER operand
 - INCLUDE directive 156
- message
 - displaying with HCPCONSL macro 9, 22, 55
 - translating into another language 51
- message examples, notation used in xv
- message repository, CP
 - advantages of 51
 - calling local 4, 25
 - commenting 209
 - control line 209
 - controlling local 4, 41
 - creating local 4, 51
 - example 52
 - message record format 209
 - overview 209
 - using local 4
- MODATTR operand
 - MDLATENT macro 181
- mode, addressing
 - addressing mode 14
 - choosing for CP exit routine 13
 - entry point attributes 14
 - register styles 13
 - translation mode 14
- MODID operand
 - MDLATENT macro 183
- modifications, CP
 - migrating to CP Exits 5
- modify
 - CP commands 4
 - CP load list 213
 - CP mods to CP exits 9
 - diagnose codes 4
 - LOGON command operands 109
 - return code from parsing LOGON command 112
 - separator page 145
 - system initialization options 101
 - VIMDBK
 - after creation/initialization 114
- MODIFY CMD command 39
- MODIFY CMD statement 39
- MODIFY COMMAND command 39
- MODIFY COMMAND statement 39
- MODIFY DIAGNOSE command 40
- MODIFY DIAGNOSE statement 40
- MODIFY EXIT command 41
- MODIFY EXIT statement 41
- module, CP
 - attribute list
 - defining 181
 - marking the beginning of 186
 - marking the end of 187
 - restrictions for CP routines 185
 - attributes

- module, CP (*continued*)
 - attributes (*continued*)
 - addressing mode 14
 - register style 13
 - translation mode 14
 - defining attribute list 181
 - packaging 35
 - writing 35
- MP operand
 - MDLATENT macro 182
 - OPTIONS directive 159
 - when to specify 34

N

- new
 - alias
 - defining 49
 - commands
 - using CP parser 63, 189
 - diagnose codes 44
 - message repository 209
 - version 46
- NEXT operand
 - HCPXSERV macro 171
- NOCONTROL operand
 - OPTIONS directive 159
 - when to specify 34
- NODELAY operand
 - when to specify 33
- NOLET operand
 - OPTIONS directive 159
 - when to specify 34
- NOLOCK operand
 - OPTIONS directive 159
- NOMP operand
 - OPTIONS directive 159
 - when to specify 34
- NONE operand
 - HCPXSERV macro 171
 - MDLATENT macro 184
- NONMP operand
 - MDLATENT macro 182
 - OPTIONS directive 159
 - when to specify 34
- NONRSTD operand
 - MDLATENT macro 185
- notation used in message and response examples xv
- NOTRACE operand
 - MDLATENT macro 184, 185
- nucleus
 - CP
 - ordered module list 213

O

- OPTIONS directive 158
- options, system initialization
 - changing 101
 - examining 101
 - providing 97
 - when prompted for 99, 103
- output data

output data (*continued*)
 receiving from CP exit routine [12](#)
overview
 CP message repository [209](#)
 CP parser [63](#), [189](#)

P

parameter list
 AOFPARM [167](#)
 AONPARM [166](#)
 defining for CP exit point [170](#)
 dynamic exit point [149](#)
 HCPACUOF entry point [167](#)
 HCPACUON entry point [166](#)
parser, CP
 creating a syntax definition [64](#)
 HCPCFDEF macro [63](#), [190](#)
 HCPDOSYN macro [63](#), [191](#)
 HCPSTDEF macro [63](#), [193](#)
 HCPTKDEF macro [63](#), [196](#)
 overview [63](#), [189](#)
 syntax descriptions [63](#)
 token conversion types [63](#), [198](#)
PERMANENT operand
 OPTIONS directive [159](#)
 when to specify [33](#)
PLIST operand
 HCPXSERV macro [172](#)
PLISTBLD operand
 HCPXSERV macro [172](#)
prevent
 include loop [156](#)
 system initialization loop
 for CP Exit 00E0 [100](#)
 for CP Exit 00E1 [104](#)
printer separator page exit
 CP Exit 4400 [133](#)
 CP Exit 4401 [135](#)
 CP Exit 4402 [137](#)
 CP Exit 4403 [138](#)
 CP Exit 4404 [140](#)
 CP Exit 4405 [142](#)
 CP Exit 4406 [143](#)
printer trailer page exit
 CP Exit 4407 [145](#)
privilege class [44](#)
process
 another file when loading code [156](#)
 LOGOFF command before completion [119](#)
 LOGON command before completion [116](#)
provide
 input data to CP exit routine [12](#)
 system initialization options [97](#)

Q

QUERY CPCMDS command [39](#), [81](#)
QUERY CPLANGLIST command [82](#)
QUERY CPXLOAD command [81](#)
QUERY DIAGNOSE command [40](#), [81](#)
QUERY EXITS command [81](#)
QUERY ICLNAME command [82](#)

QUERY UNRESOLVED command [81](#)
QWORDS operand
 HCPXSERV macro [173](#)

R

read
 another file when loading code [156](#)
receive
 output data from CP exit routine [12](#)
record format
 messages [209](#)
redefine
 CP commands [4](#)
 diagnose codes [4](#)
register styles [13](#)
reject
 LOGON command [109](#)
release
 lock on component ID [170](#)
relinquish
 lock on component ID [170](#)
remove
 lock on component ID [170](#)
rename
 external symbol, temporarily [152](#)
replace
 console input data [109](#)
 LOGON command operands [109](#)
repository, CP message
 advantages of [51](#)
 calling local [4](#), [25](#)
 commenting [209](#)
 control line [209](#)
 controlling local [4](#), [41](#)
 creating local [4](#), [51](#)
 example [52](#)
 message record format [209](#)
 overview [209](#)
 using local [4](#)
RESIDENT operand
 MDLATENT macro [182](#)
resolution, address constant
 definition [31](#)
 effect on
 CPXLOAD of prohibiting [29](#)
 CPXLOAD when unresolved [31](#)
 SYSGEN when unresolved [31](#)
 when resolved by CPXLOAD [29](#)
response examples, notation used in [xv](#)
restriction
 attributes for CP routines [185](#)
 CPXLOAD, specifying CP routine attributes [185](#)
return code
 changing
 from parsed LOGON command [112](#)
 examining
 from parsed LOGON command [112](#)
 standard [94](#)
routine, dynamically-loaded
 choosing addressing mode for [13](#)
 controlling [39](#)
 creating [11](#)
 definition [1](#)

- routine, dynamically-loaded (*continued*)
 - deleting external symbol for, temporarily [152](#)
 - design issues [11](#)
 - including another file when loading [156](#)
 - providing input data to [12](#)
 - receiving output data from [12](#)
 - renaming external symbol for, temporarily [152](#)
 - restrictions for CP routine attributes [185](#)
 - setting defaults for loading [158](#)
- RSTD operand
 - MDLATENT macro [185](#)

S

- save area, HCPSAVBK
 - contents [166](#)
 - used by CP accounting exit [166](#)
- select
 - addressing mode for CP exit routine [13](#)
- separator page exit for printers
 - CP Exit 4400 [133](#)
 - CP Exit 4401 [135](#)
 - CP Exit 4402 [137](#)
 - CP Exit 4403 [138](#)
 - CP Exit 4404 [140](#)
 - CP Exit 4405 [142](#)
 - CP Exit 4406 [143](#)
- service, CP
 - available for CP Exit 00E0 [100](#)
 - available for CP Exit 00E1 [104](#)
- set
 - defaults for loading CP routines [158](#)
 - lock on component ID [170](#)
- SHORTREG operand
 - MDLATENT macro [182](#)
- SHORTREGENT operand
 - MDLATENT macro [184](#)
- size
 - increasing CSECT [154](#)
- start
 - CP module attribute list, marking [186](#)
- statement
 - ASSOCIATE
 - EXIT [41](#)
 - MESSAGES [41](#)
 - MSGs [41](#)
 - CPXLOAD
 - loading CP command routines [39](#)
 - loading CP exit routines [40](#)
 - loading diagnose code routines [40](#)
 - loading message repositories [41](#)
 - parameter explanations [33](#)
 - restrictions for CP routine attributes [185](#)
 - DEFINE
 - ALIAS [39](#)
 - CMD [39](#)
 - COMMAND [39](#)
 - DIAGNOSE [40](#)
 - EXIT [40](#)
 - DISABLE
 - CMD [39](#)
 - COMMAND [39](#)
 - DIAGNOSE [40](#)
 - EXITS [40](#)

- statement (*continued*)
 - ENABLE
 - CMD [39](#)
 - COMMAND [39](#)
 - DIAGNOSE [40](#)
 - EXITS [40](#)
 - MODIFY
 - CMD [39](#)
 - COMMAND [39](#)
 - DIAGNOSE [40](#)
 - EXIT [41](#)
- STATIC operand
 - MDLATENT macro [184](#)
- storage addresses [2](#)
- summary
 - assigned CP exit number ranges [89](#)
 - CP exit macros [169](#)
 - CPXLOAD directives [151](#)
 - HCPXSERV register contents
 - after execution [176](#)
 - during execution [174](#)
 - IBM-defined CP exit point range assignments [89](#)
 - IBM-defined CP exit points [90](#)
 - valid HCPXSERV action/parameter combinations [174](#)
- supply
 - input data to CP exit routine [12](#)
 - system initialization options [97](#)
- symbol
 - locating [87](#)
- symbol, external
 - deleting temporarily [152](#)
 - renaming temporarily [152](#)
- syntax diagrams, how to read [xiii](#)
- SYSIOMON address for I/O monitor user exit [293](#)
- system authorization
 - examining CP commands after [105](#)
- system configuration file
 - ASSOCIATE
 - EXIT [41](#)
 - MESSAGES [41](#)
 - MSGs [41](#)
 - CPXLOAD statement
 - loading CP command routines [39](#)
 - loading CP exit routines [40](#)
 - loading diagnose code routines [40](#)
 - loading message repositories [41](#)
 - parameter explanations [33](#)
 - restrictions for CP routine attributes [185](#)
 - DEFINE statement
 - ALIAS [39](#)
 - CMD [39](#)
 - COMMAND [39](#)
 - DIAGNOSE [40](#)
 - EXIT [40](#)
 - DISABLE statement
 - CMD [39](#)
 - COMMAND [39](#)
 - DIAGNOSE [40](#)
 - EXITS [40](#)
 - ENABLE statement
 - CMD [39](#)
 - COMMAND [39](#)
 - DIAGNOSE [40](#)
 - EXITS [40](#)

system configuration file (*continued*)

MODIFY statement

CMD [39](#)

COMMAND [39](#), [40](#)

EXIT [41](#)

system execution space

dynamic area

definition [2](#)

loading code into [3](#), [33](#)

unloading code from [3](#)

layout [2](#)

system initialization options

changing [101](#)

examining [101](#)

providing [97](#)

when prompted for [99](#), [103](#)

T

table

assigned CP exit number ranges [89](#)

CP exit macros [169](#)

CPXLOAD directives [151](#)

HCPXSERV register contents

after execution [176](#)

during execution [174](#)

IBM-defined CP exit point range assignments [89](#)

IBM-defined CP exit points (summary) [90](#)

valid HCPXSERV action/parameter combinations [174](#)

table entry block, command

changing after authorization processing [105](#)

creating with COMMD macro [7](#)

TEMPORARY operand

OPTIONS directive [159](#)

when to specify [33](#)

test

CP exit points [170](#)

TEST operand

HCPXSERV macro [171](#)

TMODEAR operand

MDLATENT macro [183](#)

TMODEARENT operand

MDLATENT macro [184](#)

TMODEINT operand

MDLATENT macro [183](#)

TMODESTD operand

MDLATENT macro [183](#)

TMODESTDENT operand

MDLATENT macro [184](#)

TMODEVAR operand

MDLATENT macro [183](#)

TMODEVARENT operand

MDLATENT macro [184](#)

trace

CP Exit 00E0 [101](#)

CP Exit 00E1 [105](#)

TRACE operand

MDLATENT macro [184](#), [185](#)

trademarks [296](#)

trailer page exit for printers

CP Exit 4407 [145](#)

translate

messages into another language [51](#)

translate-and-test table

translate-and-test table (*continued*)

defining in parameter list for CP exit routine [172](#)

translation mode [14](#)

TRTTABLE operand

HCPXSERV macro [172](#)

U

unload

dynamically

from SXS [3](#)

UNLOCKEXCLUSIVE operand

HCPXSERV macro [173](#)

UNLOCKSHARED operand

HCPXSERV macro [173](#)

update

CP load list [213](#)

use

local CP message repositories [4](#)

user exit to monitor real I/O requests [293](#)

V

VMDBK

after creation/initialization

changing [114](#)

examining [114](#)

W

warning

calling entry point after loading CP routines [158](#)

CONTROL operand of OPTIONS directive [158](#)

increasing CSECT space beyond program limits [154](#)

preventing include loop [156](#)

using CHANGE and EXPAND directives together [154](#)



Product Number: 5741-A09

Printed in USA

SC24-6269-74

