z/VM
7.3

CMS Application Development Guide for Assembler

SC24-6257-73

Note:

Before you use this information and the product it supports, read the information in “Notices” on page 503.

This edition applies to version 7, release 3 of IBM® z/VM® (product number 5741-A09) and to all subsequent releases and modifications until otherwise indicated in new editions.

Last updated: 2022-09-07

Contents

Figures

Tables

About This Document

This document describes how you can use the facilities of the IBM® z/VM® Conversational Monitor System (z/VM CMS) to develop and manage assembler application programs.

Intended Audience

This information is for assembler language application and system programmers who want to develop programs in the ESA/390™, ESA/XC, z/Architecture®, or z/XC environment.

This information assumes that the reader has experience writing and running assembler language programs and that the reader is familiar with z/VM CMS.

It also assumes that the reader is somewhat familiar with ESA/390™, ESA/XC, z/Architecture, or z/XC architectures.

- ESA/390 architecture is defined in *IBM Enterprise Systems Architecture/390 Principles of Operation*, SA22-7201.
- ESA/XC architecture is defined in *z/VM: ESA/XC Principles of Operation*.
- z/Architecture is defined in *IBM z/Architecture Principles of Operation*, SA22-7832.
- z/XC architecture is defined in *z/VM: z/Architecture Extended Configuration (z/XC) Principles of Operation*.

It also assumes the reader is somewhat familiar with ESA/XC architecture, which is defined in *z/VM: ESA/XC Principles of Operation*, and z/XC architecture, which is defined in *z/VM: z/Architecture Extended Configuration (z/XC) Principles of Operation*.

Where to Find More Information

For information about related publications, see the "Bibliography" on page 507.

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 Send Your Comments to IBM

We appreciate your input on this publication. Feel free to comment on the clarity, accuracy, and completeness of the information or give us any other feedback that you might have.

To send us your comments, go to z/VM Reader's Comment Form (https://www.ibm.com/systems/campaignmail/z/zvm/zvm-comments) and complete the form.

If You Have a Technical Problem

Do not use the feedback method. Instead, do one of the following:

- Contact your IBM service representative.
- Contact IBM technical support.
- See IBM: z/VM Support Resources (https://www.ibm.com/vm/service).
- Go to IBM Support Portal (https://www.ibm.com/support/entry/portal/Overview).

Summary of Changes for z/VM: CMS Application Development Guide for Assembler

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.

SC24-6257-73, 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.

SC24-6257-02, 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 added. 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 topics is updated:

- Part 1, " Introduction," on page 1 and subtopics, including several in Chapter 3, "Architecture," on page 23

SC24-6257-01, z/VM 7.2 (September 2020)

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

Updates reflect the removal of KANJI language files from base z/VM components. The only currently supported languages are American English and uppercase English.

SC24-6257-00, z/VM 7.1 (September 2018)

This edition supports the general availability of z/VM 7.1.

Part 1. Introduction

This part of the document introduces the CMS programming environment. Part 1, " Introduction," on page 1 includes the following chapters:

- Chapter 1, "The CMS Programming Interface," on page 3 provides an overview of the CMS programming interface. The following topics are discussed:
  - The three interface groups
  - The purpose of each group
  - The actual macros, functions, and services that each group provides
- Chapter 2, "CMS Operating Characteristics," on page 15 summarizes how CMS works and how the behavior characteristics of CMS can influence the way you design your application programs.
- Chapter 3, "Architecture," on page 23 describes how your assembler language application programs are affected by the differences between System/370™ architecture and later architectures. The differences in storage addressing, PSWs, I/O handling, and assembler language instructions are described.

Chapter 1. The CMS Programming Interface

This chapter:

- Introduces and defines the CMS programming interface.
- Describes the different interface groups, the intent of each group, and the facilities that make up the group.

Overview of the CMS Programming Interface

The CMS programming interface is a way for you to get CMS to do work for you. It is made up of three groups: the CMS preferred interface group, the CMS compatibility group, and the OS/MVS and DOS/VSE group. To understand the concept of the CMS programming interface groups, you should first understand the virtual machine environments that CMS runs in.

CMS Virtual Machine Environments

z/VM provides two versions of CMS:

ESA/390 CMS (generally referred to simply as CMS)
CMS runs in the following virtual machine architectures:

ESA/390 (ESA or XA virtual machine)
An ESA virtual machine simulates IBM Enterprise Systems Architecture/390 (ESA/390), which is a superset of IBM Enterprise Systems Architecture/370 (ESA/370), which is a superset of IBM System/370 Extended Architecture (370-XA). The XA virtual machine designation is supported for compatibility; an XA virtual machine is functionally equivalent to an ESA virtual machine.

ESA/XC (XC virtual machine)
An XC virtual machine processes according to IBM Enterprise Systems Architecture/Extended Configuration (ESA/XC), which is an architecture unique to virtual machines. Although XC virtual machines run with dynamic address translation off, they can take advantage of a subset of dynamic address translation architectural features, and in particular, data spaces.

z/Architecture CMS (z/CMS)
z/CMS runs in the following virtual machine architectures:

z/Architecture (ESA or XA virtual machine)
z/Architecture uses 31-bit addressing mode in an ESA, XA, or Z virtual machine. CMS programs can use z/Architecture instructions, including those that operate on 64-bit registers, while permitting existing ESA/390 architecture CMS programs to continue to function without change.

When z/CMS is IPLed in an ESA/390 (ESA or XA) virtual machine, z/CMS switches the virtual machine to z/Architecture mode and thereafter executes in z/Architecture mode.

z/XC (XC virtual machine)
A z/XC guest uses VM Data Spaces with z/Architecture in the same way that an ESA/XC guest uses VM Data Spaces with Enterprise Systems Architecture. CMS applications that run in z/Architecture can use multiple address spaces. z/CMS can use VM Data Spaces for accessing Shared File System (SFS) Directory Control (DIRCONTROL) directories. z/XC supports programs that employ z/Architecture instructions and registers (within the limits of z/CMS support) and programs that exploit data spaces in the same CMS session.

When z/CMS is IPLed in an XC virtual machine, z/CMS switches the virtual machine to z/XC mode and thereafter executes in z/XC mode.

Unless otherwise indicated, "CMS" means either version, and descriptions of CMS functions apply to both CMS and z/CMS. For information on the differences between z/CMS and CMS and how to use z/CMS, see *z/VM: CMS Planning and Administration*.

The virtual machine mode is defined by using the MACHINE or GLOBALOPTS directory statement, the CP SET MACHINE command, or the Systems Management application programming interfaces.

Note: CP does not support System/370 architecture (370 mode) virtual machines. However, the 370 Accommodation Facility allows many CMS applications written for System/370 virtual machines to run in ESA/390 and ESA/XC virtual machines. The CP level of the 370 Accommodation Facility is activated with the CP SET 370ACCOM command. The CMS level of the 370 Accommodation Facility is activated with the CMS SET CMS370AC command. In addition, although the 370 option of the GENMOD command is not supported, modules generated with the 370 option can be run in an ESA/390 or ESA/XC virtual machine by issuing the CMS SET GEN370 OFF command. For more information about the 370 Accommodation Facility, see *z/VM: CP Programming Services*. See *z/VM: CP Commands and Utilities Reference* for information on the SET 370ACCOM command and see *z/VM: CMS Commands and Utilities Reference* for information on the SET CMS370AC and SET GEN370 commands.

The relationships between virtual machines and processor architectures are summarized in Table 1 on page 4.

Table 1. Comparison of CMS Virtual Machine Architectures

CMS Version	Virtual Machine Architecture Mode[1]	Virtual Machine Architecture	Addressing Scheme	Addressable Primary Storage	Addressable Data Space[2]
CMS	ESA, XA[3]	ESA/390	31-bit	2047 MB	2 GB per data space[4]
CMS	XC	ESA/XC	31-bit and access registers	2047 MB	2 GB per data space
z/CMS	ESA, XA[3], Z	z/Architecture[5]	31-bit[6]	2047 MB[7]	2 GB per data space[4]
z/CMS	XC	z/XC[8]	31-bit[6] and access registers	2047 MB[7]	2 GB per data space

Notes:

1. Architecture mode is set by using the SET MACHINE command and the MACHINE statement of the directory entry.
2. Multiple data spaces are possible.
3. The XA designation is supported for compatibility. An XA virtual machine is functionally equivalent to an ESA virtual machine.
4. Data spaces can be read but cannot be modified. Data spaces can be modified only in virtual machines that run in XC architectur mode.
5. When z/CMS is IPLed in an ESA/390 virtual machine, z/CMS switches the virtual machine to z/Architecture and thereafter executes in z/Architecture mode.
6. Although z/CMS does not exploit or explicitly support 64-bit addressing mode, programs running on z/CMS can enter 64-bit addressing mode.
7. Although z/CMS does not directly exploit storage above 2047 MB, z/CMS can be IPLed in a virtual machine with more than 2 GB of storage and allows programs to use storage above 2 GB.
8. When z/CMS is IPLed in an XC virtual machine, z/CMS switches the virtual machine to z/XC and thereafter executes in z/XC mode.

CMS Programming Interface Groups

Understanding the content and purpose of each group within the programming interface can help you select the CMS facility that is most appropriate for your program.

CMS preferred interface group

These macros, routines, and functions make up the heart of the CMS programming interface. They provide you with a means of making program calls, managing storage, performing I/O, handling interrupts, and processing abends. They run in ESA, XA, and XC virtual machines and they support[1] 24-bit and 31-bit addressing. They help you avoid architecture-constrained facilities like I/O instructions, they reduce your need to reference CMS internal data areas and control blocks, and they make it easier for you to develop programs that are portable across architectures.

IBM encourages you to use the preferred interface when writing your CMS application programs. "CMS Preferred Interface" on page 6 lists the macros, functions, and routines that make up the preferred interface and summarizes the services they perform. For detailed information on the preferred macros and functions, see the *z/VM: CMS Macros and Functions Reference*. For more information on the preferred routines, see "CMS Preferred Routines" on page 9.

CMS compatibility group

These are macros, functions, and services that CMS maintains for compatibility with previous releases. Existing programs that use interfaces in the compatibility group can run in 24-bit addressing mode in ESA, XA, and XC virtual machines. Compatibility group interfaces may cause unpredictable results in 31-bit addressing mode. Calling macros from this interface group in access register mode in an XC virtual machine causes an abend with a CMS abend code ofX'1CD'.

For new programs, IBM recommends that you use interfaces in the preferred group rather than interfaces in the compatibility group. "CMS Compatibility Interface" on page 10 lists the compatibility group interfaces and their suggested replacements. For detailed information on the macros and functions that make up the compatibility group, see the *z/VM: CMS Macros and Functions Reference*.

OS/MVS and DOS/VSE group

These are macros also provided by the OS/MVS and DOS/VSE operating systems. CMS supports these macros to make it easier to run on CMS programs developed for OS/MVS or DOS/VSE. The OS/MVS and DOS/VSE group consists of the following sub-groups:

1. Simulated OS/MVS macros — CMS simulates the function of the OS/MVS macros so you can use them in your programs. While these macros provide some portability between VM and OS/MVS systems, the CMS simulation of these macros is not necessarily the same as the current MVS support. CMS simulates only a selected subset of OS/MVS macros and, because of operational differences between VM and MVS, macros that are supported may work differently between the two systems. Chapter 22, "Developing OS/MVS Programs under CMS," on page 317 summarizes some of the differences between the CMS and OS/MVS support of OS/MVS macros. For complete information on how to use OS/MVS macros, you may need to refer to OS/MVS publications.

   For CMS application programs, IBM recommends that you use macros in the preferred group rather than OS/MVS macros. "Simulated OS/MVS Macros" on page 11 lists the simulated OS/MVS macros.
2. Nonsimulated OS/MVS macros — You can use these macros to develop and compile programs for execution on MVS systems; however, because CMS does not simulate these macros, programs that use them will not run on CMS. For a list of these macros, see "OS/MVS Macros for Assembly Only" on page 337.
3. DOS/VSE macros — These are DOS/VSE macros that CMS simulates. Note that the CMS simulation of these macros is not necessarily the same as the current DOS/VSE support. **For CMS application programs, IBM recommends that you use macros in the preferred group rather than DOS/VSE macros.**

   For information on these macros see Chapter 24, "Developing VSE Programs under CMS," on page 403.

Note: CMS macros, control blocks, and functions that are not part of the defined programming interface are considered internal to CMS. They should not be used by programs other than CMS.

[1] Only CMS levels prior to CMS Level 12 can run in a 370 virtual machine.

CMS Preferred Interface

This section describes the macros, routines, and functions that make up the CMS preferred interface group.

CMS Preferred Macros

The following table defines the assembler macros in the CMS preferred interface. It lists the macros that are part of the preferred interface and describes what function the macro provides. All of the macros in the preferred interface run in ESA, XA, and XC virtual machines; in XC and XA virtual machines they support 24-bit and 31-bit addressing.

The preferred macros are described in this book and in the *z/VM: CMS Macros and Functions Reference*.

Note: OpenExtensions macros are also considered part of the preferred interface. These macros, which provide mapping for OpenExtensions callable services, are not listed in Table 2 on page 6 or described in this book. For more information, see the *z/VM: OpenExtensions Callable Services Reference*.

Table 2. CMS Preferred Macros

Macro	Function
ABNEXIT	Sets or clears abend exit routines.
AMODESW	Switches or sets a program's addressing mode (AMODE) and provides an architecture-independent replacement to assembler language linkage instructions (BAL, BALR, BSM, BASSM, BAS, BASR).
ANCHOR	Allows setting, querying, and clearing a fullword that can be used to save the address of a program's data between calls.
APPLMSG	Accesses and displays messages from a message repository file.
BATLIMIT	Is a table of processor, punch, and printer limits for CMS batch jobs.
CMSCALL	Calls other programs. Use it as a replacement for SVC 202.
CMSCVT	Communications vector table.
CMSDEV	Places information about device characteristics in a user-provided buffer.
CMSECVT	Extended communications vector table.
CMSIUCV	For IUCV: establish or end IUCV communications with another program or with CP. For APPC/VM: establish or end an APPC/VM conversation, resolve a symbolic destination name, or query a workunit associated with a conversation.
CMSLEVEL	Maps the value of the feature or licensed program returned by the QUERY CMSLEVEL command. **Note:** You can also use the DMSQEFL CSL routine to return information about the level of CMS to a program.
CMSRET	Program return mechanism. Use it with CMSCALL.
CMSSTACK	Places data on the program stack. Use it as a replacement for the ATTN function.
CMSSTOR	Obtains and releases free storage. Use it as a replacement for the DMSFREE and DMSFRET macros.
COMPSWT	Sets the compiler switch on or off.
CONSOLE	Performs full-screen I/O services.
CQYSECT	Maps console path and device information to the buffer a user specifies on the CONSOLE OPEN or CONSOLE QUERY macro.

Table 2. CMS Preferred Macros (continued)

Macro	**Function**
CSFCB	Maps the data referenced by the fourth word in the Extended Plist for the CMS subcommand interface when inhibiting implicit recursion of execs.
CSLENTRY	Provides the entry logic for a callable services library (CSL) routine.
CSLEXIT	Provides the exit logic for a CSL routine.
CSLFPI	Allows an application to invoke a CSL routine using a fast path.
CSLGETP	Allows a CSL routine to get information about passed parameters.
CSRCMPSC	Calls Data Compression Services to compress and expand data.
CSRYCMPD	Maps compression and expansion dictionary entries.
CSRYCMPS	Maps the CBLOCK parameter list for calls to Data Compression Services.
DIRBUFF	Maps the records returned by a Get Directory request.
DMSABEXP	Used with the DCB abend exit to map the parameter list.
DMSABN	Abends a virtual machine.
DMSFST	Maps the file status table for a given file.
DMSJNEPL	Maps the parameter list used by the DMSJNE exit routine.
DMSSDWA	Maps the area pointed to by register 1 upon entry to an ABNEXIT routine.
DMSSTATE	Conditions preferred-group macros so that access-register mode toleration code is expanded at assembly time.
ENABLE	Enables and disables the PSW interrupt mask.
EPLIST	Maps the extended parameter list passed in register 0.
EXITBUFF	Generates a DSECT for the general data buffer that SFS provides for the File Space Usage and User Storage Group Full exits.
EXSBUFF	Maps the records returned by an Exist request for a file or a directory.
EXTUAREA	Contains external interrupt status information.
EXTXCTL	Resumes execution of code that was suspended by a X'2603' external interrupt after this interrupt occurred.
FPERROR	Maps the file pool extended error information returned in the *wuerror* parameter of CSL routines.
FSCB	Sets up a file system control block.
FSCBD	Maps the file system control block.
FSCLOSE	Closes a file.
FSERASE	Erases a file.
FSOPEN	Opens a file.
FSPOINT	Resets the write and/or read pointers for a file.
FSREAD	Reads a record from a file.
FSSTATE	Checks for an existing file.
FSTD	Maps the FST area.

Table 2. CMS Preferred Macros (continued)

Macro	**Function**
FSWRITE	Writes a record into a disk file.
GETSID	Stores a device's subchannel identification number (SID) in register 1. GETSID is required in XC and XA mode only.
HNDEXT	Defines handler routines for external interrupts.
HNDINT	Defines handler routines for I/O interrupts.
HNDIO	Defines handler routines for I/O interrupts and returns device-related information.
HNDIUCV	Initializes or ends a program's IUCV or APPC/VM environment.
HNDSVC	Defines handler routines for SVCs.
HSVCSAVE	Maps the save area passed to interrupt handlers defined by HNDSVC.
IMMBLOK	Maps the immediate command name block.
IMMCMD	Declares, clears, and queries immediate commands.
INTBLOK	Maps the I/O information that the HNDIO macro returns.
LABSECT	Maps control block for tape label processing.
LANGBLK	Generates a language control block for an application.
LINERD	Reads a line of input from the terminal.
LINEWRT	Writes a line of output to the terminal.
LRDD	Used with the LINERD macro to map the LINERD descriptors for multiple inputs.
LWRD	Used with the LINEWRT macro to map the LINEWRT descriptors for multiple outputs.
NUCEXT	Declares, clears, and queries nucleus extensions.
NUCON	Generates a mapping of the ACMSCVT, ADEVTAB, AEXEC, NUCXFRES, and USERLVL fields of the NUCON macro. **Note:** These are the only fields in NUCON that are supported as programming interfaces.
PARSECMD	Parses command arguments.
PARSERCB	Generates a parser control block DSECT.
PARSERUF	Generates a mapping for the user token validation parameter function control block.
PRINTL	Prints one or more lines on the printer.
PUNCHC	Punches a card.
PVCENTRY	Maps the parser validation code table.
RDCARD	Reads a card from the reader.
RDTAPE	Reads a record from tape.
REGEQU	Generates symbolic register equates.
REXEXIT	Invokes and maintains a list of user specified global exits for REXX programs.

Table 2. CMS Preferred Macros (continued)

Macro	Function
RXITDEF	Assigns correct values to the symbols used for the exit routine function and subfunction codes.
RXITPARM	Maps the parameter list used to pass information between the language processor and an exit routine.
SCAN	Creates tokenized and extended parameter lists from input data.
SCBLOCK	Maps the subcommand block.
SEGMENT	Manages saved segments and segment spaces.
SGMTEXIT	Maps the SGMTEXIT control block.
SHVBLOCK	Maps the shared variable block.
SUBCOM	Defines, clears, and queries subcommand environments.
SUBPOOL	Manages free storage subpools.
TAPECTL	Positions a tape.
TAPESL	Processes standard HDR1 and EOF1 tape labels.
TRANTBL	Generates a DSECT mapping of system translation tables.
TVISECT	Generates a DSECT mapping for a nucleus extension module named DMSTVI.
USERSAVE	Maps control block for call status information.
USERSECT	An 18-fullword scratch area for user-defined purposes.
VOLSECT	Maps control block for tape label processing - used when more than 16 volume IDs are specified by the user.
WAITD	Suspends program execution until the next interrupt occurs for the specified device.
WAITECB	Suspends program execution until the specified event or events occur.
WAITT	Suspends program execution until all pending terminal I/O has completed.
WRTAPE	Writes a record to tape.
WUERROR	Maps the work unit extended error information returned in the *wuerror* parameter of CSL routines.

CMS Preferred Routines

The routines in the CMS preferred interface group are documented in the following books:

- *z/VM: CMS Callable Services Reference* describes routines that perform various general programming tasks, such as:
  - File pool and minidisk file I/O
  - File pool administration
  - Accessing REXX variables
  - Extract/Replace
  - Manipulating the CMS program stack
  - Resource recovery
  - Using VM data spaces

  - Error checking and debugging.
- *z/VM: CMS Application Multitasking* describes routines that perform multitasking and related programming tasks.
- *z/VM: OpenExtensions Callable Services Reference* describes routines that manipulate Byte File System (BFS) data.
- *z/VM: OpenExtensions Advanced Application Programming Tools* describes routines that provide application program portability across UNIX® operating system platforms.

See the *z/VM: CMS Macros and Functions Reference* for more information on the macros and functions in the CMS preferred interface.

CMS Preferred Functions

The following table defines the CMS functions in the CMS preferred interface group. All of the functions in the preferred interface run in ESA, XA, and XC virtual machines; in XC and XA virtual machines they support 24-bit and 31-bit addressing. For more information on these functions, see *z/VM: CMS Macros and Functions Reference*.

Table 3. CMS Preferred Functions

Function	Description
DISKID	Obtains information on the physical organization of a reserved minidisk.
DMSSEQ	Counts the number of logical lines in the terminal input buffer.
LANGADD	Adds a LANGBLK to the language block chain.
LANGFIND	Gets the address of an application's language control block.

CMS Compatibility Interface

Table 4 on page 10 and Table 5 on page 11 list the macros and functions that CMS supports for compatibility only and suggests replacements when applicable. Existing programs can continue to use compatibility interfaces in programs that do not support 31-bit addressing.

Note:

1. The SVC 202 instruction is also considered part of the compatibility group. The CMSCALL macro is its suggested replacement.
2. The DMSEXS and DMSKEY macros allow users to modify CMS internal data areas; their use is not encouraged.

CMS Compatibility Macros and Suggested Replacements

The compatibility macros are listed with suggested replacements; for a full description of them see the *z/VM: CMS Macros and Functions Reference*.

The table below lists the macros that z/VM CMS supports for compatibility only and suggests replacements when applicable. See the z/VM: CMS Macros and Functions Reference for further information.

Table 4. CMS Compatibility Macros and Suggested Replacements

Macro	Suggested Replacement
DISPW	Use the CONSOLE macro to perform full-screen I/O and the LINEWRT and LINERD macros to perform line mode I/O.
DMSEXS	None—its use is not encouraged.
DMSFREE	CMSSTOR macro

Table 4. CMS Compatibility Macros and Suggested Replacements (continued)

Macro	Suggested Replacement
DMSFRES	No replacement required; CMS performs the function internally.
DMSFRET	CMSSTOR macro
DMSKEY	None—its use is not encouraged.
LINEDIT	APPLMSG macro
RDTERM	LINERD macro
SCAN system service	SCAN macro
STRINIT	By default, CMS treats the STRINIT macro as a no-op. In VM/SP Release 5 and earlier releases, CMS released GETMAIN storage at command end and respected the STRINIT macro. This change should not affect your programs unless they depend on the ability to invoke a program through an SVC 202 or CMSCALL, have the program obtain free storage (through GETMAIN), and then return the storage to the original invoker. If necessary, you can use the SET STORECLR command to retain GETMAIN storage until end-of-command and enable STRINIT. If possible, new programs should use the CMSSTOR macro rather than GETMAIN to obtain free storage.
TEOVEXIT	None.
WRTERM	LINEWRT macro. Unlike WRTERM, the LINEWRT macro does not allow you to specify text on the macro call itself (you have to specify the text in a buffer). You can use the APPLMSG macro to specify text on the macro call and display it at a terminal.

CMS Compatibility Functions and Suggested Replacements

The compatibility functions are listed with suggested replacements; for a full description of them see the *z/VM: CMS Macros and Functions Reference*.

Table 5. CMS Functions and Suggested Replacements

Function	Suggested Replacement
ATTN	CMSSTACK macro, or StackWrite routine
NUCEXT	NUCEXT macro
SUBCOM	SUBCOM macro
TODACCNT	None
WAITRD	LINERD macro, or StackRead routine

Simulated OS/MVS Macros

For a list of supported parameters for each macro, see "OS/MVS Macros That CMS Simulates" on page 319.

The table below lists the Simulated OS/MVS macros. For further information, see the z/VM: CMS Macros and Functions Reference and the z/VM: CMS Application Development Guide for Assembler.

[2] The DEVTYPE interface will not return valid track or cylinder details that can be used for DASD space calculations. It is intended only to give access to default device characteristics. If detailed real DASD device characteristics are needed, see CP DIAGNOSE code X'210' in *z/VM: CP Programming Services* or the CMS DEVTYPE command in *z/VM: CMS Commands and Utilities Reference*.

Table 6. Simulated OS/MVS Macros

Macro	Function
ABEND	Terminates processing with user-specified completion and reason codes.
ATTACH	Passes control to another program at a new task level.
BLDL	Builds a directory list for a partitioned data set.
BSP	Backs up a record on a tape or disk.
BUILDRCD	Causes a buffer pool and a record area to be constructed.
CALL	Transfers control to a control section at a specified entry.
CHAP	No-op.
CHECK	Verifies READ/WRITE completion.
CHKPT	No-op.
CLOSE	Completes and secures I/O processing on a DCB.
CLOSE TYPE=T	Temporarily closes and deactivates the file.
CNTRL	No-op.
DCB	Constructs a data control block.
DCBD	Generates a DSECT for a data control block.
DELETE	Deletes a loaded program.
DEQ	No-op.
DETACH	No-op.
DEVTYPE[2]	Obtains device-type physical characteristics.
ENQ	No-op.
ESPIE	Sets up handlers for program interrupts. The caller can be in either 24-bit or 31-bit addressing mode.
ESTAE	Sets up abend exit routines.
EXCP	Executes a channel program for graphic access method (GAM).
EXTRACT	No-op.
FEOV	Forces an EOV condition on a tape or DASD file.
FIND	Locates a member of a partitioned data set.
FREEBUF	Returns a buffer to the DCB buffer pool.
FREEDBUF	Releases a simulated BDAM buffer.
FREEMAIN	Releases user-acquired storage.
FREEPOOL	Releases the DCB buffer pool.
GET	Reads system-blocked data (QSAM).
GETBUF	Acquires DCB buffer storage.
GETMAIN	Acquires user storage.
GETPOOL	Constructs a buffer pool for a DCB.
IDENTIFY	Adds an entry name to a loaded program.

Table 6. Simulated OS/MVS Macros (continued)

Macro	Function
IHAEPIE	EPIE work area mapping macro.
IHASDWA	Mapping macro for the system diagnostic work area used in ESTAE.
IHAVRA	Mapping macro for the system diagnostic work area variable recording area.
LINK	Passes control to another program at the same task level and returns to the calling program.
LOAD	Reads a program into storage.
NOTE	Manages data set positioning.
OPEN	Prepares a DCB for I/O processing.
OPEN TYPE=J	Prepares a DCB for I/O processing after an RDJFCB has been issued.
PGLOAD	No-op.
PGOUT	No-op.
PGRLSE	No-op.
PGSER	No-op.
POINT	Manages data set positioning.
POST	Signals event completion.
PUT	Writes system-blocked data (QSAM).
PUTX	Returns the updated record to the data set from which it was read.
RDJFCB	Obtains information from FILEDEF command about an OS/MVS data set.
READ	Reads a physical input record (BSAM, BDAM, BPAM).
RELSE	No-op.
RETURN	Returns from a called program.
SAVE	Saves program registers.
SETRP	Makes requests for recovery from an ESTAE/ESTAI exit.
SNAP	Dumps specified areas of storage.
SPIE	Sets up an exit to be given control under user selected program interrupts. The caller must be in 24-bit addressing mode.
SPLEVEL	Sets System/370 or 370-XA macro expansion.
STAE	Sets up an abend exit routine in a 370 virtual machine.
STAX	Sets or cancels user exit for terminal attention interrupts.
STIMER	Sets the timer interval and the timer exit routine.
STIMERM	Sets, tests, or cancels multiple timer intervals and the timer exit routines.
STOW	Updates partitioned dataset directories.
SYNADAF	Provides SYNAD analysis function.
SYNADRLS	Releases SYNADAF message and save areas.

Table 6. Simulated OS/MVS Macros (continued)

Macro	Function
SYSSTATE	Conditions preferred-group macros so that access-register mode toleration code is expanded at assembly time.
TCLEARQ	Clears terminal input queue.
TGET/TPUT	Reads or writes a terminal line.
TIME	Gets the time of day.
TTIMER	Tests or cancels the timer.
WAIT	Waits for one or more events.
WRITE	Writes a physical record (BSAM, BDAM, BPAM).
WTO/WTOR	Writes a message to the operator's terminal.
XCTL	Passes control to another program at the same task level and does not return to the calling program.
XDAP	Reads or writes direct access volumes.

Chapter 2. CMS Operating Characteristics

This chapter provides an overview of:

- CMS command search order
- CMS command processing
- Program boundaries
- CMS macro libraries.

Overview of CMS Operating Characteristics

CMS is a command-driven system; for example, you enter a command and CMS executes it. The command you enter can be a CP command, a CMS command (a command that is part of the CMS system), or it can be the name of a user-written application program (a program written by you, your local system programmer, your favorite software house, and so on).

User-written programs are generally in the form of modules or execs. Modules are programs written in and compiled (translated) into assembler language. Execs are programs written in either the REXX, EXEC 2, or EXEC language.

CMS also contains assembler language macros. These macros provide a means to dynamically access CMS services. CMS macros are contained in two macro libraries: DMSGPI MACLIB and DMSOM MACLIB. For more information on these libraries, see "Using Macro Libraries" on page 21.

CMS Command Search Order

When you enter a command in the CMS environment, CMS uses the search order described below to locate it. When the command is found, CMS stops its search and executes the command. The search order is:

1. Search for an exec with the specified command name:

   a. Search for an exec in storage. If an exec with this name is found, CMS determines whether the exec has a USER, SYSTEM, or SHARED attribute. If the exec has the USER or SYSTEM attribute, it is executed.

      If the exec has the SHARED attribute, the INSTSEG setting of the SET command is checked. When INSTSEG is ON, all accessed directories and minidisks are searched for an exec with that name. (To find a file in a directory, read authority is required on both the file and the directory.) If an exec is found, the file mode is compared to the file mode of the CMS installation saved segment. If the file mode of the saved segment is equal to or higher (closer to A) than the file mode of the directory or minidisk, then the exec on the saved segment is executed. Otherwise, the exec in the directory or on the minidisk is executed. However, if the exec is in a directory and the file is locked, the execution will fail with an error message.

   b. Search the table of active (open files for a file with the specified command name and a file type of EXEC. If more than one open file is found, the one opened first is used.

   c. Search for a file with the specified command name and a filetype of EXEC on any currently accessed disk or directory, using the standard CMS search order (A through Z).

      To find a file in a directory, read authority is required for both the file and the directory. If the file is in a directory and the file is locked, the processing fails with an error message.

2. Search for a translation or synonym of the specified command name. If found, search for an exec with the valid translation or synonym by repeating Step 1.
3. Search for a module with the specified command name:

   a. Search for a nucleus extension module.
   b. Search for a module in the transient area.
   c. Search for a nucleus-resident module.
   d. Search the table of active (open) files for a file with the specified command name and a file type of MODULE. If more than one open file is found, the one opened first is used.
   e. Search for a file with the specified command name and a file type of MODULE on any currently accessed disk or directory, using the standard CMS search order (A through Z).

      To find a file in a directory, read authority is required for both the file and the directory. If the file is in a directory and the file is locked, the processing fails with an error message.
4. Search for a translation or synonym of the specified command name. If found, search for a module with the valid translation or synonym by repeating Step 3.

If the command is not known to CMS (that is, all of the above fails), it is passed to CP.

When CMS searches for a translation or synonym (as in steps 2 and 4), the translation and synonym tables are searched in the following order:

1. User National Language Translation Table
2. System National Language Translation Table
3. User National Language Translation Synonym Table
4. System National Language Translation Synonym Table
5. CMS User Synonym Table
6. CMS System Synonym Table.

See the SET TRANSLATE command for information on the tables in steps 1 to 4. See the SYNONYM command for information on the tables in steps 5 and 6. For information on the preferred file types, see the *z/VM: CMS Application Development Guide*.

CMS Runs in Supervisor State

CMS executes in virtual supervisor state; so do applications you run under CMS. At the same time, CP is running in real problem state. This means that your virtual machine and the programs you run under CMS, can issue input/output and other privileged instructions. CP intercepts these instructions and simulates the functions of them for your virtual machine. For more information on the privileged instruction set, see *z/VM: CPI Communications User's Guide*.

How CMS Command Processing Works

The purpose of writing an application program is to perform some piece of work. To perform work, an application program acquires and manages various resources. When the application program is complete, the resources need to be *cleaned up*. While this is a very simplistic definition of application programming, there is an important point to be made: to write an application program for CMS, you have to understand how and when program resources are cleaned up.

Explicitly Releasing Resources

It is considered good programming practice for a program to explicitly clean up resources it creates. To acquire resources from CMS, you can use the CMS macro interface. For nearly all resources you can use a CMS macro to acquire, a CMS macro also exists that lets you explicitly release the resource. For example, to acquire free storage, you can use CMSSTOR OBTAIN. To release free storage, you can use CMSSTOR RELEASE. To create an interrupt handler, you can use HNDIO SET. To delete the interrupt handler, you can use HNDIO CLR.

Letting CMS Clean Up After You

For various reasons, not all programs clean up after themselves. To understand how and when CMS releases various resources that you do not explicitly release, it helps to understand the relationship between the CMS command loop, SVC levels, and abend processing. These topics are discussed in the following sections. (See "Saving Resources across Boundaries" on page 20 for a list of reasons for not explicitly releasing resources.)

When you use the CMS GENMOD command to create a MODULE file, you can indicate whether you want a module to be cleaned from storage. Use the CLEAN option if you want a module to be cleaned from storage at the end of SVC processing. Specify NOCLEAN if you want a module to remain in storage until end-of-command (Ready;).

When a relocatable module is continually executed from within a command cycle, your virtual machine storage can be exhausted. To prevent exhaustion of storage, only those relocatable modules that must remain until end-of-command should be created with the NOCLEAN option. A nonrelocatable module remains in storage until another nonrelocatable module replaces it or until end-of-command.

The CMS Command Loop

When CMS is ready for you to enter a command, it is in what is called a command loop. To invoke a command, type the name of the command, module, or exec and press enter. CMS calls the CMS routines that locate and transfer control to the module or exec representing the command. When the command completes, CMS returns control to the command loop and displays a ready message (Ready;). The point where CMS returns control is called end-of-command. The command loop cycle continues as long as you enter commands.

SVC Levels

While a command is running, it can call upon services from CMS, CP, program products, or other modules and execs. When one program calls another program, it can use an SVC 202 instruction or CMSCALL macro to pass control. In turn, whatever was called by the command may call other services. Each program called by an SVC 202 instruction or CMSCALL macro is said to be at a different SVC level.

For example, assume you enter PROGA (the name of a user-written module), PROGA calls a program named PROGB, and PROGB calls a program named PROGC. SVC 202 or CMSCALL can be issued to call each successive program.

```
User enters command
                  |
                  SVC level1
                  |
                  PROGA
                      |
                      SVC level2
                      |
                      PROGB
                          |
                          SVC level3
                          |
                          PROGC
```

As each successive program completes, CMS returns to the program that called it. This point is called end-of-SVC or SVC 202/CMSCALL termination. When PROGC completes CMS returns to PROGB, when PROGB completes CMS returns to PROGA, and when PROGA completes CMS returns to the command loop.

```
             End-of-command (Ready;)  CMS waits for another command.
             |
             SVC level1
             |
         PROGA
         |
         SVC level2
         |
     PROGB
```

```
      |
      SVC level3
      |
  PROGC
```

Note: To request supervisor assisted linkage, you should issue the CMSCALL macro. The SVC 202 instruction continues to work but only from below the 16MB line. It is also worth noting that in addition to CMSCALL or SVC 202 calls, your programs can make direct branches to other programs and CMS services. For more information, see Chapter 4, "Program Invocation - Direct Branch Linkage," on page 35 and Chapter 5, "Program Invocation - Supervisor Assisted Linkage," on page 39.

Abend Processing

CMS abnormally terminates a program when the program issues the CMS DMSABN macro or the AbnormalEnd CSL call, when a program issues the OS/MVS ABEND macro, or when it (CMS) detects a condition that might degrade the system or destroy data. You can use CMS facilities to define exit routines to try to recover from abends. If no user exit routines exist or if your user exit routine cannot recover, CMS terminates the exit routine and begins abend processing. During abend processing, CMS reclaims resources allocated while the program was running, terminates the various SVC levels, and returns control back to the command loop. At this point you can issue another command.

Determining When CMS Reclaims Resources

To summarize, there are five ways that programs implicitly release resources:

1. At module creation — When you use the CMS GENMOD command to create a MODULE file, you can use the CLEAN/NOCLEAN option to indicate whether you want a module to be cleaned from storage. CLEAN is the default for relocatable modules. NOCLEAN is the default for nonrelocatable modules.
2. At SVC 202/CMSCALL termination — By default, CMS reclaims resources at SVC 202/CMSCALL termination (end-of-SVC), the point when CMS returns control from the called routine to the caller.
3. At end-of-command — If CMS does not reclaim resources at SVC 202/CMSCALL termination, it reclaims them at end-of-command (Ready;).
4. Abend processing — When a program terminates abnormally, CMS reclaims resources allocated while the program was running, terminates the various SVC levels, and returns control back to the command loop.
5. At logoff — when a user issues the LOGOFF command, all resources associated with the virtual machine are released.

SVC 202/CMSCALL termination and end-of-command can be thought of as program boundaries. Programs on one side of the boundary need not worry about resources created by programs on the other side.

If the program is a multitasking program, it is run in a new process. At end-of-SVC, the process is deleted and process-affiliated resources are cleaned up.

Figure 1 on page 20 illustrates the boundary relationship CMS environment[3]. The numbers in the drawing indicate the sequence of events:

Step 1
User begins a z/VM session by logging on.

Step 2
Command cycle begins when `cmd 1` is issued.

Step 3
The `cmd 1` calls a program and starts SVC level 1.

Step 4
The program called in Step 3 calls a program, starting SVC level 2.

[3] If you are from an MVS background, a VM session (LOGON to LOGOFF) is comparable to an MVS *JOB*, a CMS command cycle is comparable to an MVS *JOB STEP*, and a CMS SVC level is comparable to an MVS task or RB level.

Step 5
The program called in Step 4 calls a program, starting SVC level 3.

Step 6
End-of-SVC level 3 cycle, CMS reclaims resources.

Step 7
End-of-SVC level 2 cycle, CMS reclaims resources.

Step 8
Program at SVC level 1 calls another program.

Step 9
End-of-SVC level 2 cycle, CMS reclaims resources.

Step 10
End-of-SVC level 1 cycle, CMS reclaims resources.

Step 11
End command cycle for `cmd 1`. If CMS does not reclaim resources at end-of-SVC, it reclaims them at end-of-command.

Step 12
A new command is issued, `cmd 2`.

Step 13
A program is called by `cmd 2`.

Step 14
End-of-SVC level 1 cycle.

Step 15
End command cycle for `cmd 2`.

Step 16
Logging off ends the VM session. All resources associated with the virtual machine are released.

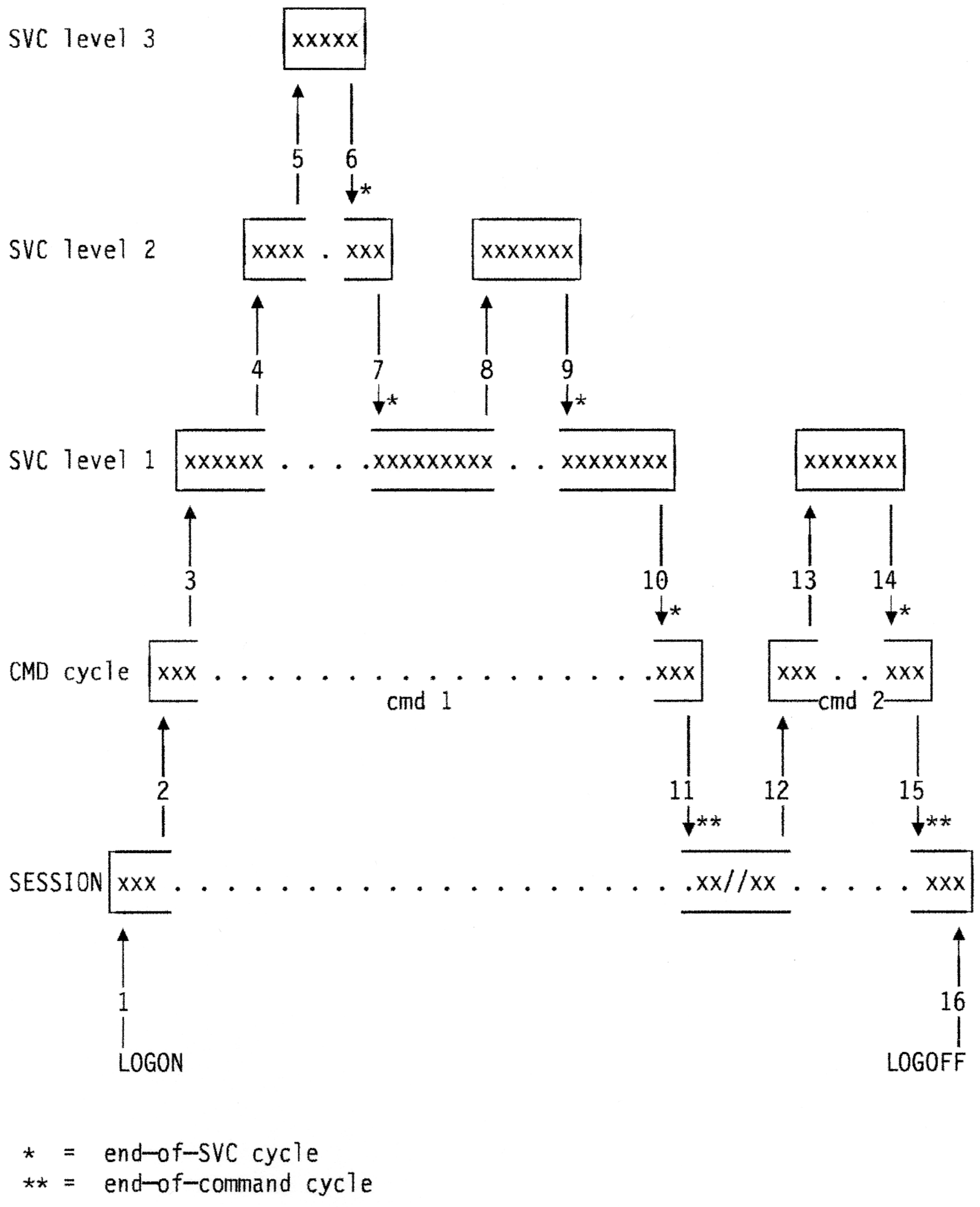

SVC level 3
xxxxx
5
6
*
SVC level 2
xxxx . xxx
xxxxxxx
4
7
*
8
9
*
SVC level 1
xxxxxx . . . .xxxxxxxxxx . . xxxxxxxx
xxxxxxx
3
10
*
13
14
*
CMD cycle
xxx xxx
cmd 1
xxx . . xxx
cmd 2
2
11
**
12
15
**
SESSION
xxx .xx//xx xxx
1
LOGON
16
LOGOFF
* = end-of-SVC cycle
** = end-of-command cycle

Figure 1. CMS Boundary Relationships

Saving Resources across Boundaries

As mentioned earlier, there are several reasons why you might want to save a resource across a boundary. For example,

- Existing programs may rely on the ability to save resources (such as GETMAIN free storage) across a boundary. If so, you can issue SET STORECLR ENDCMD. (See "Cleaning Up GETMAIN Storage" on page 338 for details.)
- You may want to save interrupt handlers across end-of-command. If so, specify the KEEP parameter when you define the interrupt handler. (See Chapter 12, "Interrupt Handling," on page 173.)
- You may want to save various resources across an abend. If so, see "What You Can Save Across a CMS Abend" on page 204.

Using Macro Libraries

Most CMS interfaces are available as CMS macros or callable services. All CMS macros that are supported as programming interfaces are organized into the following CMS macro libraries:

- DMSGPI contains most of the CMS programming interface macros. In prior releases, these macros were in DMSSP MACLIB and CMSLIB MACLIB, which no longer exist.
- DMSOM contains mostly CMS internal macros. The TEOVEXIT, IO, CMSCB, and DMSJNEPL macros are the only macros in DMSOM MACLIB that you can use as a programming interface. All other macros in DMSOM are designed for CMS internal use and should not be used as programming interfaces.

See the *z/VM: CMS Macros and Functions Reference* for a list of CMS macros that can be used by customers for programming interfaces.

To assemble a program that uses the CMS programming interface macros, you must issue the GLOBAL command specifying MACLIB DMSGPI. This macro library is usually located on the system disk. If you use any of the four CMS programming interface macros located in DMSOM, you must also specify DMSOM on the GLOBAL command.

CP macros that are supported as programming interfaces are located on the HCPGPI and HCPPSI CP macro libraries. The IUCV macro and the APPCVM macro reside on HCPGPI.

To assemble a program that uses the CP programming interface macros, you must issue the GLOBAL command specifying one or both of these macro libraries. HCPGPI MACLIB is usually located on the system disk.

See the *z/VM: CP Programming Services* for a list of CP macros that can be used by customers for programming interfaces.

Refer to "OS/MVS Macro Libraries" on page 319 for information on OS/MVS macro libraries.

Coding CMS Macros

Coding conventions for CMS macros are the same as those for all assembler language macros. If a macro statement overflows to a second line, you must use a continuation character in column 72. The continuation line must begin in column 16. No blanks may appear between operands. Incorrect coding of any macro results in assembler errors and MNOTEs.

The *z/VM: CMS Macros and Functions Reference* contains a list of the possible error conditions that may occur during the execution of CMS macros, and the associated return codes. These return codes are always placed in register 15.

The macros that produce these return codes have ERROR= operands, which allow you to specify the address of an error handling routine so that you can check for particular errors during macro processing. If an error occurs during macro processing (with the exception of CMSSTOR macro processing) and no error address is provided, execution continues at the next sequential instruction following the macro. (If an error occurs during CMSSTOR macro processing and no error address is provided, the program abends.)

How CMS Macros Work

Briefly, CMS macros provide a means to dynamically access CMS services. You issue a macro, the macro generates code to call a CMS routine, the routine performs the required function and then returns to your program. The following figure illustrates this process.

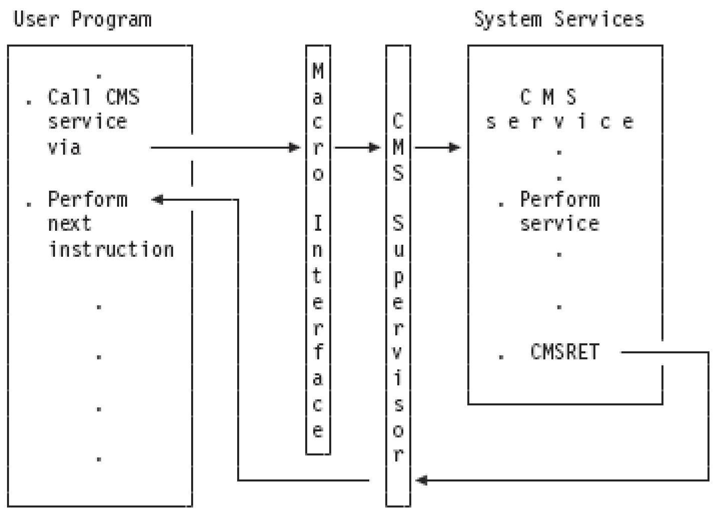


Figure 2. How CMS Macros Work

CMS Macro Formats

For many of the macros in the preferred interface group, CMS provides four macro formats:

- Standard
- List
- Complex list
- Execute.

For more information on these macro formats, see the *z/VM: CMS Macros and Functions Reference*.

Chapter 3. Architecture

The differences among ESA/390, ESA/XC, z/Architecture, and z/XC architectures that might affect application programs are described, including differences among the following elements:

- PSWs
- Registers
- Storage addressing
- I/O handling
- Assembler language instructions

Architectures are defined in other publications:

- ESA/390 architecture is defined in *IBM Enterprise Systems Architecture/390 Principles of Operation*, SA22-7201.
- ESA/XC architecture is defined in *z/VM: ESA/XC Principles of Operation*.
- z/Architecture is defined in *IBM z/Architecture Principles of Operation*, SA22-7832.
- z/XC architecture is defined in *z/VM: z/Architecture Extended Configuration (z/XC) Principles of Operation*.

ESA/390, ESA/XC, z/Architecture, and z/XC Architecture

CMS provides a programming interface to ESA/XC, ESA/XA, z/Architecture, and 370-XA architecture. To take advantage of ESA/XC or z/XC architecture, run your program in an XC virtual machine. To use ESA/390 architecture or z/Architecture, you can run your program in an ESA virtual machine. Most CMS applications that run in a 370 virtual machine can run in an ESA or ESA/XC virtual machine if you use one of the following commands:

- CP SET 370ACCOM ON
- CMS SET CMS370AC ON

In addition, modules that are generated with the 370 option of the GENMOD command can run in an ESA or ESA/XC virtual machine by issuing the CMS SET GEN370 OFF command.

For more information, see Chapter 1, "The CMS Programming Interface," on page 3.

ESA/390, ESA/XC, z/Architecture, z/XC, and System/370 PSWs

Application programs that you run under CMS operate differently in an ESA, XC, or Z virtual machine than they do in a 370 virtual machine. One reason for operational differences is the difference between the ESA, XC, or Z PSW and the System/370 PSW. The PSWs among the different architectures have the following differences:

- In the ESA/390 and ESA/XC PSWs, bit 32 specifies whether the current program runs in 24-bit or 31-bit addressing mode.
  - 24-bit addressing is indicated when bit 32 is set to 0.
  - 31-bit addressing is indicated when bit 32 is set to 1.
- In the z/Architecture and z/XC architecture PSW, bits 31 and 32 specify whether the current program runs in 24-bit, 31-bit, or 64-bit addressing mode:
  - A value of 00 indicates 24-bit addressing mode.
  - A value of 01 indicates 31-bit addressing mode.
  - A value of 11 indicates 64-bit addressing mode.

- The setting of bit 12 determines the format of the PSW:
  - Bit 12 is set to 1 for ESA/390 and ESA/XC PSWs.
  - Bit 12 is set to 0 for System/370 PSWs.
  - Bit 12 is set to 0 for z/Architecture and z/XC PSWs.
- In ESA/390, ESA/XC, z/Architecture, and z/XC, bit 6 of the PSW and control register 6 are used to mask I/O interrupts. Note that you can use the architecture-independent ENABLE macro to enable and disable your virtual machine for various types of interrupts. For more information, see "Manipulating the PSW Interrupt Mask" on page 173.

Note: z/VM supports the BC-mode, or Basic Control mode, PSW for System/370. The EC-mode, or Extended Control mode, PSW is not supported.

For more information about PSW definitions, see the following publications:

- *IBM Enterprise Systems Architecture/390 Principles of Operation*, SA22-7201
- *z/VM: ESA/XC Principles of Operation*
- *IBM z/Architecture Principles of Operation*, SA22-7832
- *z/VM: z/Architecture Extended Configuration (z/XC) Principles of Operation*

31 Bit Addressing

ESA and ESA/XC use 31 bit addressing. You can address up to 2 GB of storage. In practice, the amount of real storage you have is limited by the processor unit that you use. The amount of virtual storage you can address is limited by controls in the CP directory. CMS supports ESA, XC, and XA virtual machine storage sizes of up to 2047 MB. z/CMS supports virtual machine storage sizes of up to 128 GB. XC virtual machines provide even more storage by using multiple address spaces.

Conventions for 31-Bit Programs

Programs that run in ESA XC or XA virtual machines must follow these conventions:

- Any data passed to a 24-bit addressing mode program must reside in low storage, at an address that is less than 16 MB.
- Any data that is passed to a 31-bit addressing-mode program must reside in low storage, at an address that is less than 2 GB.
- A program must return control to a caller in the same addressing mode in which it received control. The CMSCALL, CMSRET, and AMODESW macros can handle address mode switching automatically.
- Programs must expect 31-bit addresses from 31-bit addressing mode programs and 24-bit addresses from 24-bit addressing mode programs.
- A 31-bit program must zero the high-order byte of any address it receives from a 24-bit program. Zero in the high-order byte indicates 24-bit addressing capability.
- A 64-bit program must zero the high-order word of any address it receives from a 31-bit program or the high-order 5 bytes of any address it receives from a 24-bit program.

Bimodal Addressing

CMS supports *bimodal addressing*. Bimodal addressing allows programs that run in an ESA XC or XA virtual machine to execute in 24-bit addressing mode, 31-bit addressing mode, or a combination of both addressing modes. Programs that have addressing sensitivities (that is, they are limited to 24-bit addressing) can run in an XC or XA virtual machine without being converted to 31-bit addressing.

To support bimodal addressing, CMS recognizes two program attributes: *addressing mode* (AMODE) and *residency mode* (RMODE).

Restriction: CMS does not support AMODE 64 or RMODE 64.

Addressing Mode (AMODE)

Addressing mode refers to the type of address (31-bit or 24-bit) a program expects to handle when it receives control. A program's AMODE attribute determines its addressing mode:

AMODE 24
: means a program can handle 24-bit mode addresses only. An AMODE 24 program must reside below the 16MB line.

AMODE 31
: means a program can handle 31-bit mode addresses. An AMODE 31 program can reside above or below the 16MB line.

AMODE ANY
: means you are deferring the decision to assign the program an addressing mode. There are several points in the program cycle when you can assign an AMODE or override an existing AMODE. You can also use AMODE ANY to let the program's addressing mode default to the value of the program that called it.

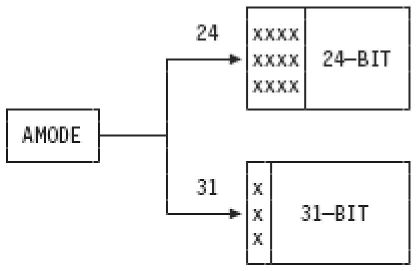


Figure 3. How CMS Interprets the AMODE Attribute

Residency Mode (RMODE)

Residency mode refers to where a program resides when CMS loads it (above or below the 16MB line). A program's RMODE attribute determines its residency mode:

RMODE 24
: means that CMS loads the program below the 16MB line.

RMODE ANY
: means that CMS loads the program above 16MB unless insufficient storage is available above the 16MB line.

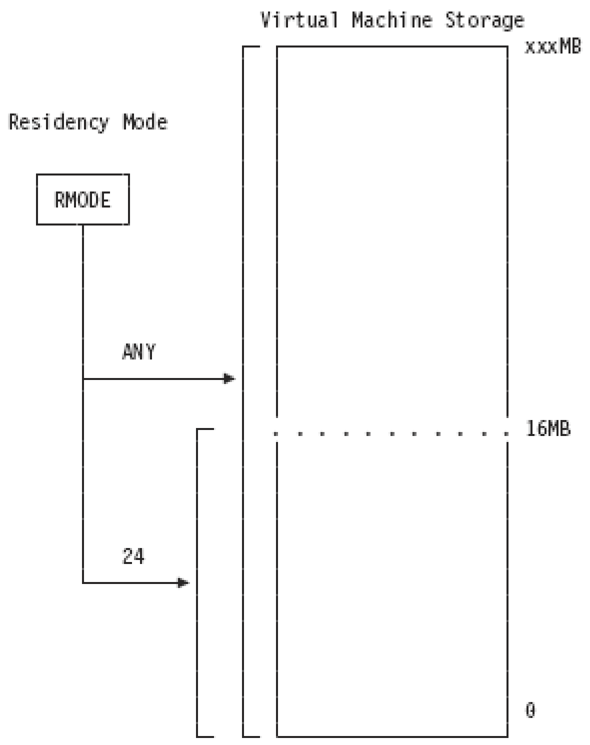


Figure 4. How RMODE Affects Where CMS Loads Programs

Setting the Addressing and Residency Modes

There are several ways you can set a program's addressing and residency modes:

Default values

If you do not specify addressing mode or residency mode, the default values are:

Specified	**Default Value**
Neither	AMODE 24, RMODE 24
AMODE 24	RMODE 24
AMODE 31	RMODE 24
AMODE ANY	RMODE 24
RMODE 24	AMODE 24
RMODE ANY	AMODE 31

Note:

1. The combination AMODE 24 and RMODE ANY is invalid.
2. The CMSCALL macro calls AMODE ANY programs in the addressing mode of the caller. The CMSCALL macro can handle address mode switching automatically.
3. If in an ESA XC or XA virtual machine you start an AMODE ANY module from the command line or from an EXEC, the module is called as AMODE 31.
4. CMS does not support AMODE 64 or RMODE 64.

AMODE and RMODE instructions

Coding the Assembler H or High Level Assembler AMODE and RMODE instructions in your assembler programs associates addressing mode and residency mode values with text files.

For example, to define a CSECT named PGMA as an AMODE 31/RMODE ANY program, you could use the AMODE and RMODE pseudo-ops as follows:

```
PGMA      CSECT
PGMA      RMODE ANY
PGMA      AMODE 31
          .
          .
          .
```

You can override these values when you use the LOAD command to load the text files into storage or when you use the GENMOD command to generate relocatable module files. Both of these commands are discussed in detail in the *z/VM: CMS Commands and Utilities Reference*. The assembler invoked by the CMS ASSEMBLE command does not support AMODE and RMODE instructions.

The format of the AMODE instruction is:

Name	Operation	Operand
Any symbol or blank	AMODE	24/31/ANY

The name field of the AMODE instruction associates an addressing mode with a CSECT in an object module. If there is a symbol in the name field of an AMODE statement, that symbol must also appear in the name field of a START, CSECT, or COM statement in the assembly.

Similarly, the name field of the RMODE statement associates the residency mode with a control section. The format of the RMODE instruction is:

Name	Operation	Operand
Any symbol or blank	RMODE	24/ANY

The RMODE and AMODE instructions can appear anywhere in the assembly. Their appearance does not initiate an unnamed CSECT. While an assembly can have more than one RMODE (or AMODE) instruction, each instruction must have a different name field.

LOAD command

You can use the AMODE, RMODE, and ORIGIN options to set the addressing mode and residency mode of the text file you load. Addressing mode and residency mode values specified on the LOAD command override any values set at assembly time. Also, note that ORIGIN and RMODE are mutually exclusive parameters; you cannot specify both on the same LOAD command.

The AMODE option lets you specify an addressing mode (24, 31, or ANY) to the program you load. The RMODE option specifies the residency mode (24 or ANY).

The ORIGIN option lets you specify an address where CMS is to load a program and, therefore, sets the residency mode implicitly. If you specify an ORIGIN address 16MB or greater, the program is assigned a residency mode of ANY. If you specify an ORIGIN address less than 16MB, the program is assigned a residency mode of 24.

GENMOD command

The AMODE and RMODE options of the GENMOD command allow you to specify AMODE and RMODE values for modules you generate.

For example, to generate a module named 31BITGUY with an addressing mode of 31 and a residency mode of ANY, enter:

```
genmod 31bitguy (amode 31 rmode any
```

To generate a module named 24BITGUY with an addressing mode of 24 and a residency mode of 24, enter:

```
genmod 24bitguy (amode 24 rmode 24
```

The AMODE option overrides any values specified at assembly time or on the LOAD command. The RMODE option also overrides any values specified at assembly time or on the LOAD command. For nonrelocatable modules, the RMODE option of the GENMOD command is ignored and the module load address is determined by the location of the text file when GENMOD is issued.

SUBCOM and NUCEXT macros

You can use the SUBCOM macro to define subcommand processors and the NUCEXT macro to define nucleus extensions. Both macros let you specify an addressing mode for the programs.

The MVS/XA Linkage Editor

If you use the MVS/XA linkage editor, the AMODE and RMODE attributes are modified at link-edit time by default values, or by values set in the PARM field of the LKED command or the linkage editor MODE control statement. By default, the linkage editor checks each CSECT of the entire load module, and sets the RMODE to the lowest mode encountered (in other words, if the linkage editor finds **any** RMODE 24 CSECTs, it assigns the entire module the value RMODE 24). The linkage editor sets the module AMODE to the AMODE of the entry point unless an AMODE is explicitly specified.

Note: For more information on addressing mode, residency mode, and the program packaging process in general, see Chapter 15, “Program Packaging,” on page 211.

Calling Other Programs

CMS provides two macros, CMSCALL and AMODESW, that you can use to call other programs and, if necessary, switch addressing modes:

- Use CMSCALL to request supervisor assisted linkage to other programs; CMSCALL automatically calls the program in the correct mode. For more information on CMSCALL, see Chapter 5, “Program Invocation - Supervisor Assisted Linkage,” on page 39.
- Use AMODESW as an architecture-independent replacement for direct branch instructions such as BAL, BALR, BAS, BASR, BSM, and BASSM. For more information on AMODESW, see Chapter 4, “Program Invocation - Direct Branch Linkage,” on page 35.

For more information on AMODESW and CMSCALL, see the *z/VM: CMS Macros and Functions Reference*.

I/O Considerations

One of the major differences between ESA/390 architecture, z/Architecture, and System/370 architecture is in the way that I/O is handled. Each architecture has a unique instruction set and a unique method of handling I/O to and from devices. Your CMS applications can complete I/O by using the CMS macro interface, DIAGNOSE instructions, or the actual ESA/390 or z/Architecture instructions.

The following sections describe the various levels of support in more detail. The level that you choose depends on the requirements of your application.

Information about the channel subsystem that is used by ESA and z/Architecture and how System/370 I/O handling differs from channel subsystem I/O handling is available in architecture publications:

- ESA/390 architecture is defined in *IBM Enterprise Systems Architecture/390 Principles of Operation*, SA22-7201.
- ESA/XC architecture is defined in *z/VM: ESA/XC Principles of Operation*.
- z/Architecture is defined in *IBM z/Architecture Principles of Operation*, SA22-7832.

- z/XC architecture is defined in *z/VM: z/Architecture Extended Configuration (z/XC) Principles of Operation*.

CMS Preferred Interface I/O Support

If you run your program in an XC or XA virtual machine, you need to issue channel subsystem I/O instructions.

The best way to handle I/O to and from devices is to use the I/O macros in the CMS preferred interface.

The following table summarizes the macros CMS provides to help you write I/O code that works in an ESA XC or XA virtual machine:

Table 7. CMS Preferred Interface I/O Macros

Function	Macros	Reference
File management	FSCB FSCBD FSCLOSE FSERASE FSOPEN FSPOINT FSREAD FSSTATE FSWRITE	Chapter 9, “CMS File System,” on page 107
Console I/O	CONSOLE APPLMSG LINERD LINEWRT	Chapter 8, “Console and Terminal I/O,” on page 83
Unit Record I/O	PRINTL PUNCHC RDCARD	Chapter 11, “Unit Record Devices and Tapes,” on page 147
Tape I/O	TAPESL TAPCTL RDTAPE WRTAPE	“Tape Handling Macros” on page 151
I/O Interrupt Handling	HNDIO WAITD WAITT	Chapter 12, “Interrupt Handling,” on page 173

Using Diagnose Codes for I/O

If you cannot use the CMS macro interface and it does not matter whether your application's I/O processing is synchronous or asynchronous, you should use the DIAGNOSE code X'A4' and DIAGNOSE code X'A8' instructions. The DIAGNOSE code X'A4' and DIAGNOSE code X'A8' instructions work in ESA, Z, or XC virtual machines, but not in 64-bit addressing mode.

I/O Instructions

As mentioned earlier, ESA/390, ESA/XC, z/Architecture, and z/XC architectures and System/370 architecture support different sets of I/O instructions. When you use I/O instructions in your program, you must make sure you use the I/O instructions appropriate for the mode of virtual machine in which the program is to run. An operation exception occurs in your program if it issues the wrong mode of I/O instruction (a System/370 I/O instruction in an ESA, XC, or XA virtual machine). Most CMS applications that are written by using System/370 I/O instructions can run in an ESA, XA, or ESA/XC (but not z/XC) virtual machine. You must use the CP SET 370ACCOM ON command or the CMS SET CMS370AC ON command.

Information on the CP SET 370ACCOM command is available in another publication. See the *z/VM: CP Commands and Utilities Reference*.

Information on how to run your 370-only CMS applications in an ESA XA or XC virtual machine is available in another publication. See the *z/VM: CP Programming Services*.

Information on the CMS SET CMS370AC ON command is available in another publication. See the *z/VM: CMS Commands and Utilities Reference*.

ESA/XC and ESA/390 architectures maintain compatibility to System/370 in the areas of CCWs, IDAWs, and channel programs. However, you must be careful when you replace System/370 I/O instructions with the ESA/XC or ESA/390 counterparts; System/370 and ESA/390 or ESA/XC condition codes do not necessarily mean the same thing. (For example, TIO condition codes need to be handled differently than TSCH condition codes.)

Table 8 on page 30 lists the I/O instructions in each architecture.

Table 8. I/O Commands of several architectures

ESA/390, ESA/XC, z/Architecture, and z/XC I/O Instructions	Mnemonic	System/370 I/O Instructions	Mnemonic
CLEAR SUBCHANNEL	CSCH	CLEAR CHANNEL	CLRCH
HALT SUBCHANNEL	HSCH	CLEAR I/O	CLRIO
MODIFY SUBCHANNEL	MSCH	HALT DEVICE	HDV
RESET CHANNEL PATH	RCHP	HALT I/O	HIO
RESUME SUBCHANNEL	RSCH	RESUME I/O	RIO
SET ADDRESS LIMIT	SAL	START I/O	SIO
SET CHANNEL MONITOR	SCHM	START I/O FAST RELEASE	SIOF
START SUBCHANNEL	SSCH	STORE CHANNEL ID	STIDC
STORE CHANNEL PATH STATUS	STCPS	TEST CHANNEL	TCH
STORE CHANNEL REPORT WORD	STCRW	TEST I/O	TIO
STORE SUBCHANNEL	STSCH		
TEST PENDING INTERRUPTION	TPI		
TEST SUBCHANNEL	TSCH		

Complete information on these and other assembler language instructions are available in another publication. See *z/VM: ESA/XC Principles of Operation*.

The GETSID Macro

The ESA/390, ESA/XC, z/Architecture, and z/XC I/O operations require that the device subchannel-identification word (SID) is in register 1. Use the GETSID macro to find the SID of a specific device name or device address and to store it in register 1.

Note: The GETSID macro is needed in ESA XC or XA virtual machines.

Example

The following sequence of instructions starts I/O to device TAP1.

```
GETSID DEVNAME='TAP1'          * Get SID of TAP1 in R1
SSCH   CCWS                    * Start I/O using the CCWs
                                 at location CCWS.
```

Assembler Instructions That Work Differently

Because of architectural differences (for example, 31-bit versus 24-bit addressing capabilities) the ESA/390, ESA/XC, z/Architecture, and z/XC assembler language instruction sets are different from the System/370 assembler language instruction set. Some System/370 instructions do not work in ESA, XC, Z, or XA virtual machines. In addition, a few ESA/XC and z/XC instructions (for example, SAC) do not work in an ESA or XA virtual machine that run CMS.

Most CMS applications that are written by using System/370 specific instructions can run in an ESA, XA, or XC virtual machine. You must use the CP SET 370ACCOM ON command or the CMS SET CMS370AC ON command.

Information on the CP SET 370ACCOM command is available in another publication. See the *z/VM: CP Commands and Utilities Reference*.

Information on how to run your 370-only CMS applications in an ESA XA or XC virtual machine is available in another publication. See the *z/VM: CP Programming Services*.

Information on the CMS SET CMS370AC ON command is available in another publication. See the *z/VM: CMS Commands and Utilities Reference*.

Instructions That Are Sensitive to Addressing Mode

Certain assembler instructions (those that use or develop addresses) operate differently according to the current addressing mode. (See *z/VM: ESA/XC Principles of Operation* for details of all ESA/XC assembler instructions.)

- **BAL, BALR** — In 24-bit addressing mode, BAL and BALR work the same way as they did in a 370 virtual machine. Before branching, they put link information into the high-order byte of the first operand register and put the return address into the remaining 3 bytes.

  In 31-bit addressing mode, BAL and BALR put the return address in bits 1-31 of the first operand register and set the high-order bit to 1 to indicate a 31-bit address. To save the program mask and condition code, use the 370-XA instruction IPM (INSERT PROGRAM MASK).
- **LA** — In 24-bit addressing mode, the LA instruction loads a 24-bit address and clears the high-order byte. In 31-bit addressing mode, LA loads a 31-bit address and clears the high-order bit.

Other instructions that have been changed to handle two types of addresses include: COMPARE LOGICAL LONG, EDIT AND MARK, MOVE LONG, and TRANSLATE AND TEST. In the decimal instruction set, EDIT AND MARK places the address in register 1 according to the addressing mode in effect at the time of execution.

Instructions That Are Not Supported

ESA/390 and ESA/XC architectures do not support the following instructions:

- INSERT STORAGE KEY
- RESET REFERENCE BIT
- SET STORAGE KEY

Using any of these instructions in an ESA, XC, or XA virtual machine causes operation exceptions in your programs. However, if you issue the CP SET 370ACCOM ON command, then ISK, RRB, and SSK are supported.

The instructions that are supported in each architecture are documented in architecture publications:

- ESA/390 architecture is defined in *IBM Enterprise Systems Architecture/390 Principles of Operation*, SA22-7201.
- ESA/XC architecture is defined in *z/VM: ESA/XC Principles of Operation*.
- z/Architecture is defined in *IBM z/Architecture Principles of Operation*, SA22-7832.
- z/XC architecture is defined in *z/VM: z/Architecture Extended Configuration (z/XC) Principles of Operation*.

Part 2. Using CMS Services

This part of the document describes how to use the CMS preferred interfaces from within your programs to dynamically access CMS services. Part 2, " Using CMS Services," on page 33 includes the following chapters:

- Chapter 4, "Program Invocation - Direct Branch Linkage," on page 35 describes how to link to other programs using direct branch linkage.
- Chapter 5, "Program Invocation - Supervisor Assisted Linkage," on page 39 describes how to link to other programs using CMS supervisor assisted linkage.
- Chapter 6, "Using Free Storage," on page 53 describes how to allocate, manage, and release free storage and free storage subpools.
- Chapter 7, "Using Saved Segments," on page 75 describes how to use physical and logical saved segments.
- Chapter 8, "Console and Terminal I/O," on page 83 describes how to use macros to do terminal and console I/O.
- Chapter 9, "CMS File System," on page 107 describes how CMS manages files.
- Chapter 10, "Using the File System Macros," on page 131 describes CMS macros you can use to manipulate CMS files.
- Chapter 11, "Unit Record Devices and Tapes," on page 147 describes how to use CMS macros to perform unit record I/O and to manage tapes.
- Chapter 12, "Interrupt Handling," on page 173 discusses how CMS interrupt handling routines can make it easier for application programs to handle interrupts.
- Chapter 13, "Nucleus Extensions and Commands," on page 191 describes how to use the NUCEXT, ANCHOR, SUBCOM, and IMMCMD macros.
- Chapter 14, "Abend Processing," on page 203 discusses CMS abend processing and abend exit routines.

Chapter 4. Program Invocation - Direct Branch Linkage

This chapter:

- Provides an overview of direct branch linkage.
- Describes considerations for using BAL/BALR for AMODE ANY programs.
- Describes how to pass control from one program to another using the AMODESW macro. Note that AMODESW is an architecture-independent alternative for the BAL, BALR, BAS, BASR, BSM, and BASSM instructions.

Overview of Direct Branch Linkage

While your program executes, it can call other routines within the same program or it can call other programs. One method for calling, or linking, to other programs is called direct branch linkage. A direct branch generates fewer instructions and is faster than supervisor assisted linkage. On the other hand, you can only use a direct branch to call programs that already reside in storage (supervisor assisted linkage can locate and load a program for you).

One way to make a direct branch is to use assembler instructions such as BAL/BALR, BAS/BASR, or BSM/BASSM. BSM and BASSM are 370-XA (and ESA/XC) instructions. Developing AMODE ANY programs using BAL/BALR for program linkage is one way to develop programs that can be independent of assembler instruction mode-sensitive behavior.

Another way to perform direct branch linkage is to use the AMODESW macro. This macro can work in a XA or XC virtual machine.

Using BAL/BALR with AMODE ANY Programs

Programs can be designed to be insensitive to the addressing mode in which they receive control. These programs are referred to as AMODE ANY programs and are designed to execute in the AMODE of the caller. You can use the BALR/BAL and BR instructions to pass control to and from AMODE ANY programs.

In developing AMODE ANY programs, you should be aware of the following considerations:

- Certain machine instructions (BAL, BALR, LA, TRT, EDMK) execute differently depending on the addressing mode in effect when they are invoked. AMODE ANY programs must not have any dependency on the mode-dependent manner in which these instructions execute. For example, the BAL and BALR instructions save the instruction length code, the condition code, and the program mask in bits 0 - 7 of the link register in 24-bit addressing mode but not in 31-bit addressing mode. If this information is required by the application, you can use the IPM (Insert Program Mask) instruction instead because it operates identically in 24-bit and 31-bit addressing modes.

  For more information on the mode-sensitive behavior of these instructions, please see *z/VM: ESA/XC Principles of Operation*.
- Programs designed to operate in this manner need to consider proper program residency (RMODE) requirements. In particular, a program executing in 24-bit addressing mode cannot branch to an AMODE ANY program which resides above the 16MB line.

Switching the Addressing Mode

Use the AMODESW macro to:

- To branch to other subroutines within a module. Programs can use the AMODESW macro as an architecture-independent alternative for the BALR or BASSM instructions.

- Switch a program's current addressing mode to 24-bit or 31-bit.
- Save the current addressing mode when switching to a new addressing mode. This is useful when (a) a program does not know the current mode, (b) wants to switch to a particular mode, and (c) eventually wants to return to the original mode.
- Determine the current addressing mode.

AMODESW Formats

The general formats of the AMODESW macro are:

- AMODESW SET — To switch addressing modes.
- AMODESW CALL — To make a subroutine call with an appropriate mode switch.
- AMODESW RETURN — To return from a subroutine.
- AMODESW QUERY — To determine the current addressing mode.

See the *z/VM: CMS Macros and Functions Reference* for the complete syntax of AMODESW.

Using AMODESW - Examples

Example 1 - Switch Addressing Mode Inline

To switch addressing mode inline, use the SET function of the AMODESW macro as follows:

```
AMODESW SET,AMODE=31    Switch to 31-bit addressing
   .
   .
   .
AMODESW SET,AMODE=24    Return to 24-bit addressing mode
```

Example 2 - Branch to a Subroutine in 31-Bit Addressing Mode

To call the subroutine named MYSUB in 31-bit addressing mode, code the AMODESW macro as follows:

```
         AMODESW CALL,AMODE=31,ADDRESS=MYSUB
           .
           .
           .
MYSUB    EQU   *
           .
           .
           .
         AMODESW RETURN
```

Example 3 - Save and Restore Addressing Mode

The following example shows how to use the SAVE parameter of AMODESW. It allows MYSUB to return to a routine without being sensitive to the addressing mode that the called routine uses.

```
MYSUB    CSECT
          .
          .
          .
* Save return register, set new mode, save caller's mode
         LR   R10,R14
         AMODESW SET,AMODE=31,SAVE=(R10)
          .
          .
          .
* Return to caller, restore addressing mode
         AMODESW RETURN,REG=(R10)
```

Example 4 - Using AMODESW as an Alternative for BALR

You can use the AMODESW CALL and RETURN mechanism wherever a BALR and BR are currently used.

To replace a BALR call where registers other than 14 and 15 are used for the linkage, use the REGS parameter on AMODESW CALL and the REG parameter on AMODESW RETURN, as in the following example:

```
         AMODESW CALL,AMODE=31,ADDRESS=MYSUB,REGS=(R9,R10)
           .
           .
           .
MYSUB    EQU   *
           .
           .
           .
         AMODESW RETURN,REG=(R9)
```

Note: Because the instructions BSM and BASSM are not available in the 370 virtual machine, AMODESW generates dual-path code (code that works in either addressing mode). If your application will be executed only in XA or XC virtual machines, you can specify the MODE=NO370 parameter. The dual-path code needed to execute in a 370 virtual machine will not be generated.

Example 5 - Set Addressing Mode According to a VCON

The following example shows how you can use AMODESW to branch to a VCON address; AMODESW can automatically set the addressing mode according to the addressing mode associated with the VCON. This technique lets you keep the calling mechanism constant and assure a proper transfer of control even if the program you call changes its AMODE definition.

When the CMS loader resolves the address of a VCON, it also sets the high-order bit of the address to reflect the AMODE of the entry point. If the entry point is AMODE 31 or AMODE ANY, the high-order bit of the VCON address is set to 1. If the entry point is AMODE 24, the high-order bit of the VCON address is set to 0. This means the address to which you want to transfer control also contains the addressing mode. Note that EXTRN/ADCON specifications are not resolved by the CMS loader to reflect the addressing mode.

```
PGM1     CSECT
PGM1     RMODE 24                Define PGM1 residency mode.
PGM1     AMODE 24                Define PGM1 addressing mode.
          .
          .
         L   R15,PGM2AD          Get the entry address of PGM2.
         AMODESW CALL            Transfer control to PGM2 in the AMODE
 *                               defined for its CSECT.  By default,
 *                               R14 is the return register, AMODESW
 *                               CALL saves the addressing mode in the
 *                               high-order bit.
          .
 PGM2AD  DC  V(PGM2)             When loader resolves external ref,
 *                               it sets high-order bit according to
 *                               the AMODE defined for its CSECT.

 -------------------------------------------------------------------

 PGM2    CSECT
 PGM2    RMODE  ANY              Define PGM2 residency mode
 PGM2    AMODE  31               Define PGM2 addressing mode
          .
          .
         AMODESW RETURN          When PGM2 completes, control returns
 *                               to PGM1 according to the AMODE in R14
 *                               (saved by AMODESW CALL).
```

Note: The above example only works when you use the HOBSET option of the CMS LOAD or INCLUDE commands. This support is available only with the CMS Loader and not with the HCPLDR or LKED commands.

Example 6 - Set Addressing Mode According to an ADCON

The technique used above may also be applied to routines using EXTRNs and ADCONs. The following example defines the AMODE of the routine by setting on or off the high-order bit of the ADCON.

```
RTN1     CSECT
         EXTRN  RTN2AD
         EXTRN  RTN3AD
          .
          .
         L   R15,=A(RTN2AD)     Load address of RTN2.
         L   R15,0(R15)
         AMODESW CALL           Go to RTN2 using AMODESW.  By default,
 *                              R14 is the return register and AMODESW
 *                              CALL saves the addressing mode in the
 *                              high-order bit.
          .
          .
         L   R15,=A(RTN3AD)     Load address of RTN3.
         L   R15,0(R15)
         AMODESW CALL           Go to RTN3 using AMODESW.  By default,
 *                              R14 is the return register and AMODESW
 *                              CALL saves the addressing mode in the
 *                              high-order bit.

 RTN2    CSECT
 RTN2    AMODE   24
         ENTRY   RTN2AD

         AMODESW RETURN         Return to caller in caller's mode
 *                              according to AMODE stored in R14 by
 *                              AMODESW call.
 RTN2AD  DC A(RTN2)             When used as an address indicates
 *                              AMODE 24 because bit 0 is 0.

 RTN3    CSECT
 RTN3    AMODE   31
         ENTRY   RTN3AD

         AMODESW RETURN         Return to caller in caller's mode
 *                              according to AMODE stored in R14 by
 *                              AMODESW call.
 RTN3AD  DC A(X'80000000'+RTN3) When used as an address indicates
 *                              AMODE 31 because bit 0 is 1.
```

Chapter 5. Program Invocation - Supervisor Assisted Linkage

This chapter:

- Provides an overview of CMS supervisor assisted linkage.
- Describes how to pass control from one program to another using the CMSCALL and CMSRET macros. The SVC 202 instruction can also be used to pass control but will only work below the 16MB line.
- Describes how to use the SCAN macro to help you construct tokenized and extended parameter lists.

Overview of CMS Supervisor Assisted Linkage

Supervisor assisted linkage, as its name implies, means that CMS helps you link to a program. When you identify the program you want to call, CMS (a) locates the program, (b) loads the program and performs other preparation as required, and (c) transfers control to the program.

To request supervisor assisted linkage, you can use the CMSCALL macro, an SVC 202 instruction, or the OS/MVS ATTACH, LINK, or XCTL macros. CMSCALL is the preferred method: it is a CMS service (as opposed to the OS/MVS macros which CMS simulates) and it works from anywhere in your program (SVC 202 works only from below the 16MB line). Also, CMSCALL:

- Always loads a new copy of the program (except for nucleus extensions), thus protecting you from obtaining an unusable copy.
- Calls programs in the correct addressing mode
- Saves the caller's registers to help assure a proper return when the called program completes.
- Provides a save area which the called program can use.
- Provides the capability of copying parameter lists from above the 16MB line to below it (this capability is used when a program above the 16MB line calls a program below it).
- Lets you specify an optional routine to handle errors that occur during macro processing.

Supervisor Assisted Linkage — An Overview

In general, a program call works as follows:

1. The calling program sets up the parameter list(s) that the called program expects. There are two basic formats of parameter lists: the tokenized parameter list and the extended parameter list. You can use the SCAN macro to instruct CMS to build correctly formatted parameter lists from data you specify.
2. The calling program issues the CMSCALL macro. (Programs that run under 16MB can use SVC 202; CMSCALL, however, is the recommended method for new programs).
3. CMS allocates a user save area. The user save area is a register save area (or work area) used by the routine that is being called. When you use CMSCALL, the user save area also contains information about the type of call being made, user flags, and some miscellaneous flags (see "USERSAVE Control Block" on page 49 for details).
4. CMS passes control to the specified program. "Register Usage" on page 47 describes how CMS sets up the registers for the called program.
5. The called program returns to the calling program. To return to its caller, a program can issue the CMSRET macro or branch to the address contained in register 14. CMS releases the save areas and restores the registers (you can use CMSRET to specify that CMS pass unchanged the contents of certain registers back to the calling program).

Figure 5 on page 40 and Figure 6 on page 40 illustrate CMSCALL register and save area usage.

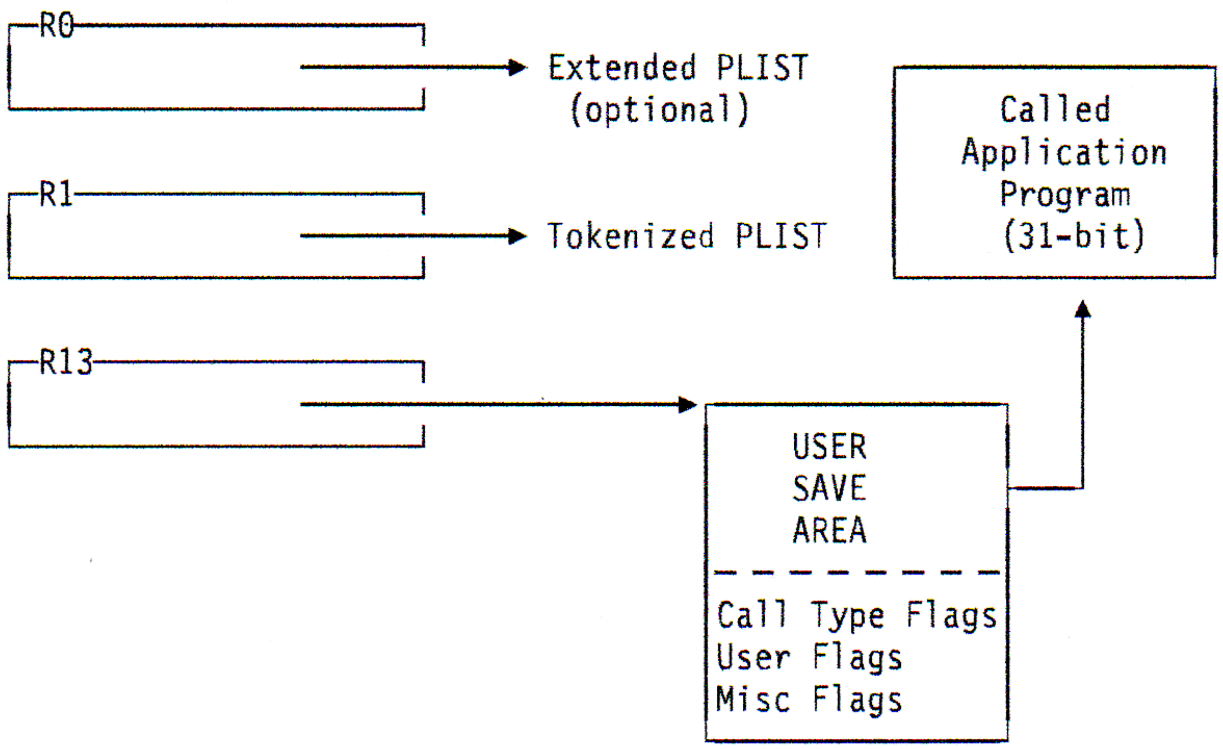


Figure 5. CMSCALL for AMODE ANY/AMODE 31 Programs

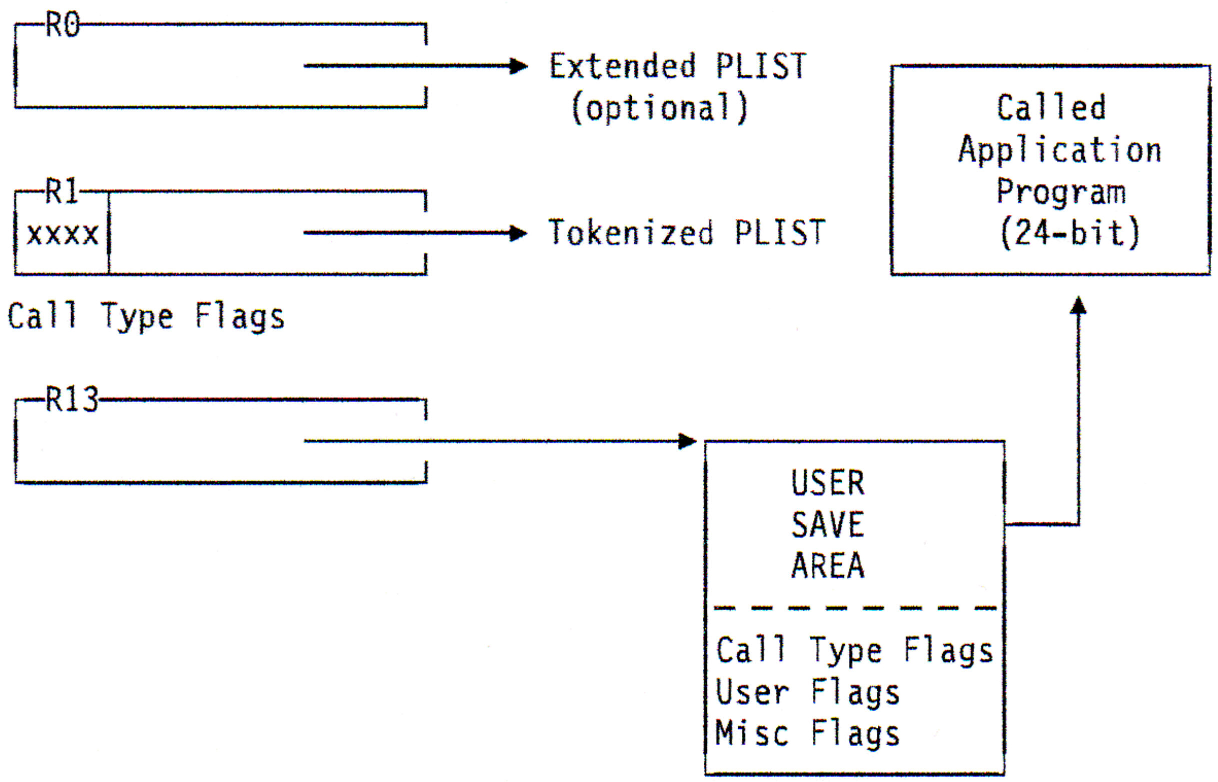


Figure 6. CMSCALL for AMODE 24 Programs

Setting Up a Parameter List

The SCAN macro provides a simple method to create tokenized and extended parameter lists from input data. SCAN also stores the address of the extended parameter list in register 0 and the tokenized parameter list in register 1; this helps you set up to use the CMSCALL macro.

Using the SCAN Macro

Before invoking the SCAN macro, your program must provide an area where CMS can construct the tokenized and extended parameter lists. The length of the area you provide depends on the number of arguments in the input data. (The input data is a string of arguments delimited by one or more blanks or a left or right parenthesis.)

Parameter List Format

The standard form of the extended and tokenized parameter lists that the SCAN macro creates is as follows:

```
.
     extended parameter list
.
DC A(CMNDNAME)   Command name
DC A(BEGARG)     Beginning of argument list
DC A(ENDARG)     End of argument list
DC F'0'          User word
DC 4F'0'         Reserved for future use
.
     tokenized parameter list
.
DC CL8'        ' Space for tokens (arguments) as required
DC CL8'        ' Space for tokens (arguments) as required
DC 8X'FF'        Fence
```

Figure 7. SCAN Macro Parameter List Format

The Extended Parameter List

The extended parameter list, the first four fullwords of the list shown in Figure 7 on page 41, contains the following information:

- The first fullword, DC A(CMNDNAME), points to the beginning of the input data, which SCAN assumes is the name of the program being invoked.
- The second fullword, DC A(BEGARG), points to the beginning of the argument string (the first nonblank character following the command name).
- The third fullword, DC A(ENDARG), points to the first byte following the end of the input data.
- The fourth fullword, DC F'0', is a user word that the SCAN macro sets to zero before returning control to the invoking program. Users can store information in this fullword before calling the program pointed to by the first word of the extended parameter list.

CMS stores the address of the extended parameter list in register 0 before control returns to the invoking program.

The Tokenized Parameter List

To create a tokenized parameter list, CMS truncates or pads each argument in the input data to 8 bytes. The first token must be the name of the program you call. In addition, CMS treats each parenthesis as an argument and it appends an 8-byte "fence" (delimiter) of X'FF' to mark the end of the tokenized parameter list.

SCAN stores a pointer to the tokenized parameter list in general register 1.

Determining Storage Needed for Parameter Lists

The storage area you provide for the SCAN macro to build its parameter lists must consist of 40 bytes **plus** 8 bytes for each argument or parenthesis in the input data. Therefore, if your input data consists of only one argument, you must provide a 48-byte storage area. For example, if you have the following input data,

```
test
```

the SCAN macro would need 48 bytes to build parameter lists, as shown below.

```
CMNDNAME DC A(CMDTKN)      CMDTKN is the address of the name of the
*                          program being invoked, in this case, 'test'
BEGARG   DC A(CMDTKN+4)    Beginning of argument list
ENDARG   DC A(CMDTKN+4)    End of argument list
         DC F'0'           User word
         DC A(0)           Address of function argument list
         DC A(0)           Address for return of function data
         DC 2F'0'          Padding
         DC CL8'test    '  Space for tokens (arguments) as required
         DC 8X'FF'         Fence
```

Figure 8. Determining Storage for PLISTs

If you supply the following argument list,

```
test(file1 file2)
```

the SCAN macro would need 80 bytes to build a parameter list, as shown below.

```
CMNDNAME DC A(CMDTKN)         CMDTKN is the address of the name of the
*                             program being invoked, in this case, 'test'
BEGARG   DC A(CMDTKN+4)       Beginning of argument list
ENDARG   DC A(CMDTKN+17)      End of argument list
         DC F'0'              User word
         DC A(0)              Address of function argument list
         DC A(0)              Address for return of function data
         DC 2F'0'             Padding
         DC CL8'file2   '     Space for tokens (arguments) as required
         DC CL8')       '     Space for tokens (arguments) as required
         DC 8X'FF'            Fence
```

Figure 9. Determining Storage for PLISTs

Note: If your buffer area is too small to contain the tokenized parameter list, CMS truncates the parameter list and returns an error code of 4 in register 15.

Determining the Number of Arguments

If you use the TEXT operand of the SCAN macro to specify the input data, you will know how many arguments it includes. If you get the data from somewhere else, you need to count the number of arguments. (An alternative is to provide storage for the maximum number of arguments you might receive).

Translation Values

Optionally, you can use the TRANS parameter of the SCAN macro to translate input data according to user defined translation values. (The SET INPUT and SET OUTPUT commands can define alternate translation values.)

When a program requests translation and a user has not defined any alternate translations, the SCAN macro translates to uppercase using the default table.

Defining Translation Values

To use translation values with SCAN, you must first use the SET INPUT command to identify the data (character or hex) you want to translate. For each character or hex value you want to translate you must enter a separate SET INPUT command (you can enter SET INPUT from an exec, from the command line, or by using CMSCALL to invoke it from a program).

For example, to translate the character ‘f’ to X'40', you could specify:

```
SET INPUT f 40
```

To translate X'17' to X'07', specify:

```
SET INPUT 17 07
```

Example

The following example shows how the SCAN macro works (assume that EXSETI issues the two SET INPUT commands):

```
SCANXMP  CSECT
         USING SCANXMP,12
         ST    14,SAVER14    Save return code
         LA    1,EXSETI
         CMSCALL             Call front-end exec to issue set input cmds
         SCAN  TEXT=(INPUT,INPLEN),BUFFER=(PLSTAREA,),TRANS=YES
         CMSCALL             COMBO must be a relocatable module
*                            or must not overlay SCANXMP
         L     14,SAVER14
         BR    14
         DS   0F
INPUT    DC   C'COMBO 14f65f32f65f'    PLIST that will be translated
         DC   X'225C175C86'            and sent to program COMBO
INPLEN   EQU  *-INPUT
SAVER14  DS    F
         DS   0D
PLSTAREA DS   CL80                     Area where SCAN builds PLIST
         DS    0F
EXSETI   DS    0F
         DC    CL8'EXEC'
         DC    CL8'SETI'
         DC    8X'FF'
         END


When COMBO is called, register 1 points to the address of the tokenized
parameter list, whose contents will be as follows:

         .
         .
         DS    0F
PLISTIN  DC    CL8'COMBO'              Name of program to be called
         DC    CL8'14'                 Char parm
         DC    CL8'65'                 Char parm
         DC    CL8'32'                 Char parm
         DC    CL8'65'                 Char parm
         DC    XL8'225E075E86404040'   Hex parm
         DC    8X'FF'                  Fence to show end of list
```

Figure 10. SCAN Macro Example

Making the Call

After you have set up the appropriate parameter list, you can use the CMSCALL macro to invoke a CMS command, CMS function, nucleus extension, nucleus resident routine, or user module. CMSCALL works from above or below the 16MB line and automatically calls the program you specify in the appropriate addressing mode.

The following sections describe how and when you can use some of the options of the CMSCALL macro.

Specifying a Call Type

Use the CALLTYP parameter of CMSCALL to specify the type of call you are making. CMSCALL stores this value in the USECTYP field of the user save area and, for 24-bit programs, in the high-order byte of register 1.

Table 9 on page 44 shows how you can specify CALLTYP and the values that get stored. (You can also specify your own call-type value.) For more information on the CMSCALL macro, see the *z/VM: CMS Macros and Functions Reference*.

Table 9. Program Invocation Call Type Options

Option	Hex Value	Description
CALLTYP=PROGRAM	X'00'	Passes a tokenized parameter list. This is the default value if you do not specify the EPLIST parameter.
CALLTYP=EPLIST	X'01'	Passes a tokenized parameter list and an extended parameter list. This call type acts as if it were invoked using ADDRESS COMMAND from REXX. This is the default value if you specify the EPLIST parameter.
CALLTYP=SUBCOM	X'02'	Uses the SUBCOM interface to make the call.
CALLTYP=NONUCXE	X'03'	Passes an extended parameter list and then, during the command search, bypasses the search of the list of nucleus extensions.
CALLTYP=NONUCXT	X'04'	Passes a tokenized parameter list and then, during the command search, bypasses the search of the list of nucleus extensions.
CALLTYP=FUNCTION	X'05'	Calls a REXX/VM interpreter function or subroutine.
CALLTYP=CMS	X'0B'	Simulates invocation from a console and passes a tokenized parameter list and an extended parameter list.

Specifying a Parameter List

If you use the SCAN macro, the addresses of a tokenized and extended parameter list are automatically stored in the appropriate registers. (Register 1 for the tokenized parameter list, register 0 for the extended parameter list.) Register 1 must contain the address of a tokenized parameter list. If you do not use SCAN to store the address of the tokenized parameter list in register 1, you can use the PLIST parameter of CMSCALL.

Specifying an Extended Parameter List

Whether you need to pass an extended parameter list to a program depends on the type of program you call. If you do need one, specify the EPLIST parameter of CMSCALL. The address of the extended parameter list should be in register 0 (the SCAN macro can do this automatically). Also, specifying EPLIST sets to one a bit (USEPLIST) in the user save area (USERSAVE).

Copying and Modifying Parameter Lists

If an AMODE 31 program calls an AMODE 24 program and the address of the parameter lists are above the 16MB line, CMS copies the contents of those lists to a location below the 16MB line. This allows an AMODE 31 program from above the 16MB line to call an AMODE 24 program below the 16MB line without worrying about the location of the parameter lists.

The COPY parameter of CMSCALL specifies whether CMS copies the parameter list. Because COPY=YES is the default value, you do not need to worry about the COPY parameter unless you specifically do not want CMS to copy the parameter lists.

When to Specify COPY=NO

Specifying COPY=NO may reduce the number of operations CMS performs if your AMODE 31 bit program frequently calls AMODE 24 programs. Here are two examples of when your program might want to specify COPY=NO:

- Your program allocates storage from below the 16MB line and stores the parameter lists there itself.
- Your program has no parameters to pass (you may just pass information in registers, or with a return code, or using the UFLAGS parameter).

Modifying Parameter Lists

If an AMODE 31 program residing above the 16MB line expects to call an AMODE 24 program which, in turn, might modify the parameter list, the AMODE 31 program can specify MODIFY=YES on the CMSCALL macro. If MODIFY=YES is specified and the AMODE 24 program modifies the parameter list, CMS copies those modifications back to the parameter list of the AMODE 31 program.

Other CMSCALL Options

The other parameters of CMSCALL are:

- UFLAGS — this is an optional one byte parameter to be stored in the USEUFLG byte of the user save area (USERSAVE). You can use UFLAGS to specify anything you like.
- FENCE — the FENCE parameter indicates whether the last token in the tokenized parameter list is the standard fence, which has a doubleword value of X'FF'. If FENCE=NO, CMSCALL copies the 68 doublewords beginning at the address of the tokenized parameter list.

  If FENCE=YES (the default), CMSCALL copies everything up to and including the fence in the tokenized parameter list. If you default to FENCE=YES then you must have a fence in your tokenized parameter list.
- ERROR — the ERROR parameter lets you specify the address where control is passed if an error occurs during CMSCALL processing.

For the complete syntax of the CMSCALL macro, see *z/VM: CMS Macros and Functions Reference*.

Call Charts

You can pass information about the type of call being made two ways:

- In the high-order byte of register 1
- In the user save area. See “USERSAVE Control Block” on page 49 for more information.

Table 10 on page 45 and Table 11 on page 46 show how CMS handles the parameter list for the CMSCALL macro and for the SVC 202 instructions.

Table 10. CMSCALL Call Chart

Parameter List Location	AMODE of Program Being Called	Action to Parameter List
Below 16MB	24	CMS copies the information specified on the CALLTYP parameter of CMSCALL into the high-order byte of general register 1. This allows CMSCALL to call a routine that has not been changed (that is, a routine that expects information about the call in the high-order byte of register 1 instead of in the user save area).

Table 10. CMSCALL Call Chart (continued)

Parameter List Location	AMODE of Program Being Called	Action to Parameter List
Above 16MB	24	Unless you code the COPY=NO parameter on the CMSCALL macro, CMS copies the parameter below the 16MB line. If you do code the COPY=NO parameter on the CMSCALL macro, the program terminates with an abend code of X'1CC'. If you do not specify COPY=NO, CMS copies the information specified on the CALLTYP parameter of CMSCALL into the high-order byte of general register 1.
Anywhere	31, ANY	Leave intact. If the caller is AMODE 24 and the callee is AMODE ANY, CMS copies the information specified on the CALLTYP parameter of CMSCALL into the high-order byte of general register 1.

Note:

1. For the CMSCALL macro, CMS always treats the address of the tokenized parameter list as a 31-bit address.
2. CMS passes register 0, which may contain the address of the extended parameter list, intact to the caller. It does not check to determine what type of address you pass unless you specify the COPY parameter on the CMSCALL macro.

Table 11. SVC 202 Call Chart

Callers Location	AMODE of Program Being Called	Action to Parameter List
Below 16MB	24	Leave intact.
Above 16MB	24, 31, ANY	Abend code X'1CA' — SVC 202 does not work from above 16MB.
Below 16MB	31, ANY	CMS stores zeros in the high-order byte of general register 1 in order to pass a 31-bit address.

Note: For SVC 202, CMS always treats the address of the tokenized parameter list as a 24-bit address.

Receiving Control

When a program receives control by CMSCALL or SVC 202, there are several ways it can obtain information.

- Parameter lists — register 1 contains the address of the tokenized parameter list. Register 0 contains the address of the extended parameter list, if one was specified.
- Register contents — see "Register Usage" on page 47 for a discussion of what the called program's registers contain.
- By interrogating the user save area — CMSCALL stores information about the call type in the user save area. For more information, see "USERSAVE Control Block" on page 49.

  For SVC 202, a program had to store call type information in the high-order byte of register 1. See "SVC 202 Call Type Values" on page 49 if you need an explanation of the various SVC 202 call type codes.
- By checking the return code — see "Return Codes" on page 50 for a list of return codes.

Register Usage

When a command or routine is called by SVC 202 or CMSCALL, the registers contain the following information:

Register
Contents

0
Points to an extended parameter list (EPLIST) if the command is called from the terminal, a REXX program, or an EXEC2 exec.

1
Points to a tokenized parameter list. If the called program is an AMODE 24 program, the high-order byte contains call-type information. If the called program is an AMODE 31 or AMODE ANY program, the high-order byte is part of the address.

Note: The user save area also contains call type information, regardless of the calling program's addressing mode (AMODE).

2
Contains a pointer to the SCBLOCK if the called program is a nucleus extension or subcommand processor; otherwise, the content of register 2 is not defined.

3-11
Content is not defined.

12
Contains the entry point address of the called program. The called program can use this address to establish addressability. Also, the high-order bit (bit 32 of the PSW) is automatically set to 0 or 1 according to the current addressing mode. Bit 32 is set to 1 for 31-bit addressing and set to 0 for 24-bit addressing.

13
Contains a pointer to a user save area that you can use to save the calling program's registers. Note, however, that saving the caller's registers is optional because CMS does it automatically.

Also note that when you use CMSCALL, the user area contains the call type flag, a user specified flag, information related to the addressing mode of the caller at the time of the CMSCALL macro, and a flag indicating whether the program linkage was done with SVC 202 or CMSCALL. The user area can be mapped using the USERSAVE macro.

14
Contains the address of the CMS program linkage routines. Your program must return control to this address when it terminates. (If you use the CMSRET macro, you can branch to this address automatically.)

15
Contains the entry point address of the called program. The called program can use this address to establish addressability. Also, the high-order bit (bit 32 of the PSW) is automatically set to 0 or 1 according to the current addressing mode. Bit 32 is set to 1 for 31-bit addressing and set to 0 for 24-bit addressing.

On return from a CMS routine, register 15 contains:

Return Code
Meaning

0
No error occurred

<0
Called routine not found

>0
Error occurred

If a CMS routine is called by an SVC 202, CMS saves and restores registers 0 through 14.

Table 12 on page 48 shows how registers are set up when a called routine is entered.

Table 12. Register Contents When Called Routine Starts

Register	CMSCALL	SVC 202	Other SVCs
0–1	Same as caller	Same as caller	Same as caller
2	See note	See note	Same as caller
3–11	Not defined	Not defined	Same as caller
12	Address of called routine	Address of called routine	Address of called routine
13	Address of user save area	Address of user save area	Address of user save area
14	Return address	Return address	Return address
15	Address of called routine	Address of called routine	Same as caller

Note: If the called routine is a nucleus extension or subcommand processor, then register 2 has the address of the SCBLOCK and the bit USESCBLK in USERSAVE is set to 1.

Interrupt Mask and Storage Key Settings

Table 13 on page 48 shows how interrupts are masked and storage keys are set up when a called routine starts.

Table 13. Settings When A Called Routine Starts

Call Mechanism	Target Program	Interrupt Masking	Storage Key Settings
CMSCALL	Nucleus resident	Disabled	System
CMSCALL	Nucleus extension module	Defined by NUCEXT macro	Defined by NUCEXT macro
CMSCALL	Transient area module	Disabled	Defined by GENMOD or SET PROTECT command
CMSCALL	User area module	Enabled	Defined by GENMOD or SET PROTECT command
SVC 202	Nucleus resident	Disabled	System
SVC 202	Nucleus extension module	Defined by NUCEXT macro	Defined by NUCEXT macro
SVC 202	Transient area module	Disabled	Defined by GENMOD or SET PROTECT command
SVC 202	User Area Module	Enabled	Defined by GENMOD or SET PROTECT command
User-defined		Disabled	User
OS-VSE	Nucleus resident	Disabled	System
OS-VSE	Transient area module	Disabled	System

Note: When a user defined SVC interrupt handler is invoked, the interrupt mask is disabled.

USERSAVE Control Block

The USERSAVE mapping macro maps the area that register 13 points to when a program uses an SVC 202 or CMSCALL to call another program. Figure 11 on page 49 shows the structure of USERSAVE.

Note: You can use the EPLIST macro to map the USECTYP field in USERSAVE.

```
         USERSAVE
USERSAVE DSECT
         DS  12D                   Reserved for the user.
USERSIZE EQU   *-USERSAVE          Size of area reserved for user.
USERINFO DS    D                   Information passed to user.
         ORG   USERINFO
USECTYP  DS    X                   Contains CALLTYP value.
USEUFLG  DS    X                   Contains UFLAGS value.
         DS   2X                   Reserved for IBM use.
USEMFLG  DS    X                   Miscellaneous bits.
USECMS   EQU   X'80'               Invoked by CMSCALL.
USEA31   EQU   X'40'               Caller's AMODE is 31.
USESCBLK EQU   X'20'               SCBLOCK is available in R2.
USEPLIST EQU   X'10'               Extended PLIST available in R0,
*                                  only valid if invoked by CMSCALL.
         DS   3X                   Reserved for IBM use.
USERSAVL EQU   (*-USERSAVE+7)/8    BLOCK LENGTH (DOUBLEWORD)
```

Figure 11. USERSAVE DSECT

SVC 202 Call Type Values

When a program called with SVC 202 gets control, it can check the USECTYP field of the user save area to determine what environment (EXEC, command line, and so forth) it was called from and if an extended parameter list is available[4].

The following values can be found in USECTYP:

Table 14. USECTYP Values

Value	Meaning	Extended PLIST Pointer in Register 0?
X'00'	The call did not originate from an EXEC file or a command typed at the terminal. (The SVC handler translates the value X'04' to X'00' before entering the called program). May be the result of CMSCALL with CALLTYP=PROGRAM.	No
X'01'	Either the call is from an EXEC 2 exec or a REXX exec when ADDRESS COMMAND is specified, or the call is an Enhanced Connectivity Facilities call (see SENDREQ in the *z/VM: CMS Macros and Functions Reference*). You can tell by checking the form of the extended parameter list, see "The Extended Parameter List" on page 41. (The SVC handler translates the value X'03' to X'01' before entering the called program). May be the result of CMSCALL with CALLTYPE=EPLIST.	Yes
X'02'	Used by SUBCOM interface. May be the result of CMSCALL with CALLTYPE=SUBCOM.	Yes[5]
X'03'	The call originated from a program and instructs CMS to bypass the list of nucleus extensions during the command search. An extended parameter list is passed. May be the result of CMSCALL with CALLTYPE=NONUCXE.	Yes

[4] For AMODE 24 programs, the call-type value is also contained in the high-order byte of register 1.

Table 14. USECTYP Values (continued)

Value	Meaning	Extended PLIST Pointer in Register 0?
X'04'	The call originated from a program and instructs CMS to bypass the list of nucleus extensions during the command search. A tokenized parameter list is passed. May be the result of CMSCALL with CALLTYPE=NONUCXT.	Yes
X'05'	Used by the REXX/VM interpreter for external function calls. May be the result of CMSCALL with CALLTYPE=FUNCTION.	Yes
X'06'	The command was invoked as an immediate command. This setting should never occur with SVC 202.	Yes
X'0B'	The command was called as a result of its name being typed at the terminal, by the CMDCALL command to invoke the command from EXEC 2, or from a REXX exec when ADDRESS CMS is specified. May be the result of CMSCALL with CALLTYPE=CMS.	Yes
X'0C'	The call is the result of a command invoked from a CMS EXEC file with &CONTROL set to something other than NOMSG or MSG.	No
X'0D'	The call is the result of a command invoked from a CMS EXEC file with &CONTROL MSG in effect (indicates that messages are to be displayed at the terminal).	No
X'0E'	The call is the result of a command invoked from a CMS EXEC file with &CONTROL NOMSG in effect.	No
X'10'	The call is the result of a command invoked by BPX1EXC. The plist passed in R1 is of the exec() type. See the *z/VM: OpenExtensions Callable Services Reference* for details.	No
X'FE'	This is an end-of-command call from the CMS console command handler (DMSINT). See the NUCEXT function in the *z/VM: CMS Macros and Functions Reference* for details.	No
X'FF'	This is a service call from abend (DMSABN) or from NUCXDROP. See the NUCEXT function in the *z/VM: CMS Macros and Functions Reference* for details.	No

Return Codes

On return from SVC 202 or CMSCALL processing, register 15 contains one of the following return codes:

0
No errors occurred.

-1
A CP command with this name was not found.

-2
An attempt was made to execute a CMS command while in CMS subset mode. This would have caused the module to be loaded in the user area.

5 There are a few SUBCOM interfaces that do not require an extended PLIST.

-3
A CMS command issued from EXEC was not found with this name, or an invalid function occurred when the SET or QUERY command was issued from EXEC with IMPCP active.

-4
The LOADMOD failed.

-5
A LOADMOD was issued in the wrong environment (for example, the module was generated by the GENMOD command with the OS option, and LOADMOD was attempted with DOS=ON specified).

-6
An attempt was made to invoke a CMS function or macro from the command line, an EXEC 2 exec with &PRESUME &SUBCOMMAND, or from a REXX exec when ADDRESS CMS is specified. The function should be invoked from a program with SVC 202 or CMSCALL with the proper parameter list.

Returning To a Program

Use the CMSRET macro to return to the caller from a program that was invoked by SVC 202, or CMSCALL. CMSRET should not be used in user defined exits or immediate commands because CMS transfers control to exits and immediate commands differently than for command invoked modules.

Consider a nucleus extension called PROG1 set up as an immediate command and you interrupt an exec with it. The immediate command exit is not considered to be a new SVC level, so if you issue a CMSRET rather than a BR 14 from PROG1, you cause the current SVC level to be removed. Because of this, the exec is terminated, along with the immediate command, since the exec was the last thing on the SSAVE chain; that is, CMS acts as though the CMSRET was from the exec.

If you have a nucleus extension that can be invoked as both a command and an immediate command, use just the BR 14, or dual path the code to test how you were entered, and proceed accordingly:

```
         .
         .
         .
         TM    USECTYP,EPLFIMMD      Entered as an Immediate Command?
         BZ    RETOUT                No, CMSRET is safe
         LM    R0,R14,0(R13)         Restore Caller's Registers
         BR    R14                   and return to IMMCMD caller
         SPACE 1
RETOUT   EQU   *                     Invoked as a nucleus extension
         CMSRET                      Return to command caller
```

This applies to:

- Immediate commands
- HNDIO
- HNDEXT
- HNDSVC
- HNDIUCV
- ABNEXIT.

The rule of thumb is if it is an EXIT, do not invoke CMSRET.

Example 1 — A Simple Return

To make a simple return, you can code the CMSRET macro with no parameters, as follows:

```
CMSRET
```

Example 2 — Setting a Return Code

By default, CMSRET returns the contents of register 15 unchanged. Therefore, you can use register 15 to send a return code to the calling program. One way to do this is to store the value you want returned in register 15 before you issue CMSRET. Another way to send a return code is to use the RC parameter of the CMSRET macro. For example, to specify a return code of 34, you could code:

```
         CMSRET RCODE
         .
         .
         .
         DS    0F
RCODE    DS    F'34'
```

Example 3 — Returning Register Contents

By default, CMSRET clears all registers (except register 15) before it returns control to the calling program. To specify that CMSRET return unchanged the contents of certain registers, use the GR and FPR parameters. For example, to return the contents of general registers 0, 3, and 5 through 8, code:

```
CMSRET GR=(0,3,5-8)
```

Chapter 6. Using Free Storage

This chapter describes how to:

- Use the CMSSTOR and SUBPOOL macros
- Manage free storage and storage subpools
- Determine how much free storage is available
- Use the STORMAP, SUBPMAP, and STDEBUG commands
- Obtain and release storage above 2 GB

Overview of Free Storage

Free storage is storage your application can allocate to dynamically create variable-sized buffers, tables, and data areas. Allocating storage while your program is running uses storage more efficiently—you can allocate storage only when you need it so your modules can be smaller.

The amount of free storage available to your applications depends on the storage size of your virtual machine, the amount of storage used by CMS, and the amount of storage used by other applications you run.

CMSSTOR and SUBPOOL Macros

CMS provides two macros, CMSSTOR and SUBPOOL, to help you allocate, manage, and release free storage and free storage subpools. CMSSTOR and SUBPOOL macros are designed to work together. Programs must issue CMSSTOR explicitly to obtain and release storage; subpool management works whether programs specify the SUBPOOL macro or not.

Use the CMSSTOR macro to allocate and release free storage. The basic variations of CMSSTOR are:

- CMSSTOR OBTAIN — Allocates free storage.
- CMSSTOR RELEASE — Releases free storage.

The basic variations of the SUBPOOL macro are:

- SUBPOOL CREATE — Creates a free storage subpool.
- SUBPOOL DELETE — Deletes the subpool from the list of active subpools and releases the subpool's storage.
- SUBPOOL RELEASE — Releases the subpool's storage but does not delete the subpool from the list of available subpools.

CMSSTOR and SUBPOOL are recommended for use in new programs or programs that you convert to exploit 31-bit addressing. Other storage macros include the CMS macros DMSFREE, DMSFRES, and DMSFRET and the OS/MVS macros GETMAIN and FREEMAIN. DMSFRES is treated as a no-op. with existing programs. You can continue to use them but only from below 16 MB. GETMAIN and FREEMAIN are OS/MVS macros that CMS simulates. Using CMSSTOR and SUBPOOL in your programs is more efficient. Also note that there are differences in the way different releases of CMS handle GETMAIN storage.

CMS Storage Layout

Storage layout is determined by the size of the virtual machine relative to the top of the nucleus, which is typically at 20 MB. The location and size of the page allocation table vary according to the size of the virtual machine. In virtual machines larger than 20 MB, the page allocation table is built at the top of virtual storage. In virtual machines smaller than or equal to 20 MB, the page allocation table extends down from whichever is less: the bottom of the nucleus or the top of virtual storage.

Figure 12 on page 54 and Figure 13 on page 54 show sample CMS storage layouts for virtual machines whose sizes are less than or equal to the top of the nucleus.

Figure 14 on page 55 shows a sample CMS storage layout for a virtual machine that extends beyond the top of the nucleus. Free storage starts above the transient program area and extends up to the limit of the virtual machine size (excluding any storage used by CMS).

For more information on the size and location of the page allocation table, see the *z/VM: CMS Planning and Administration*.

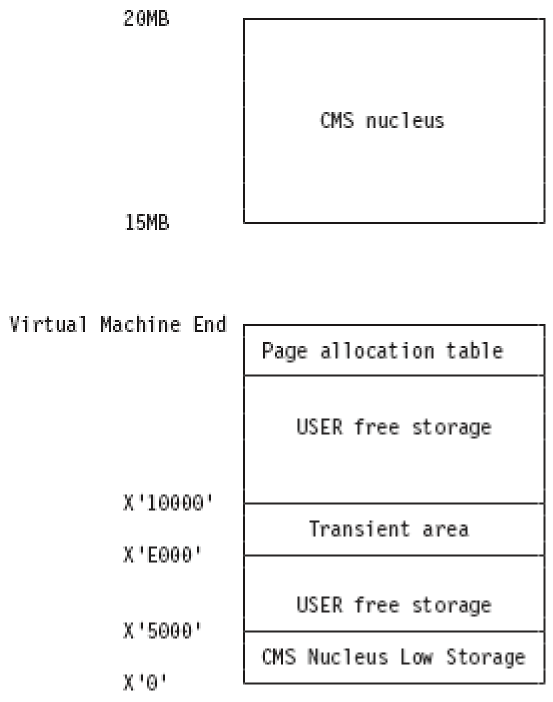


Figure 12. Storage Configuration for a Virtual Machine Less Than 15 MB

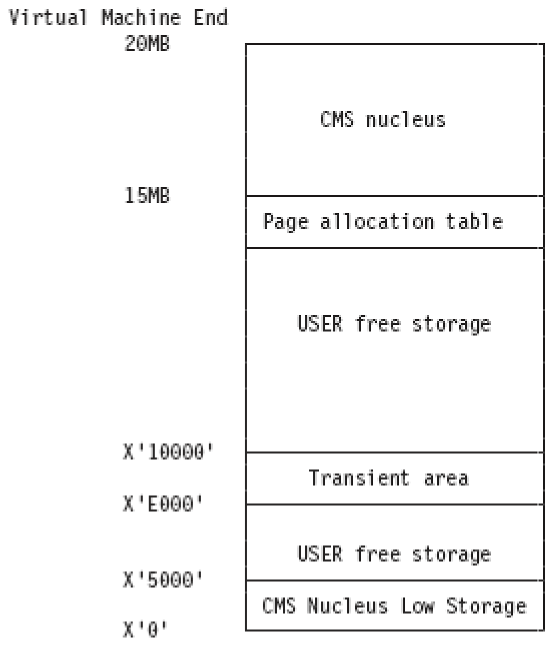


Figure 13. Storage Configuration for a Virtual Machine Equal to 20 MB

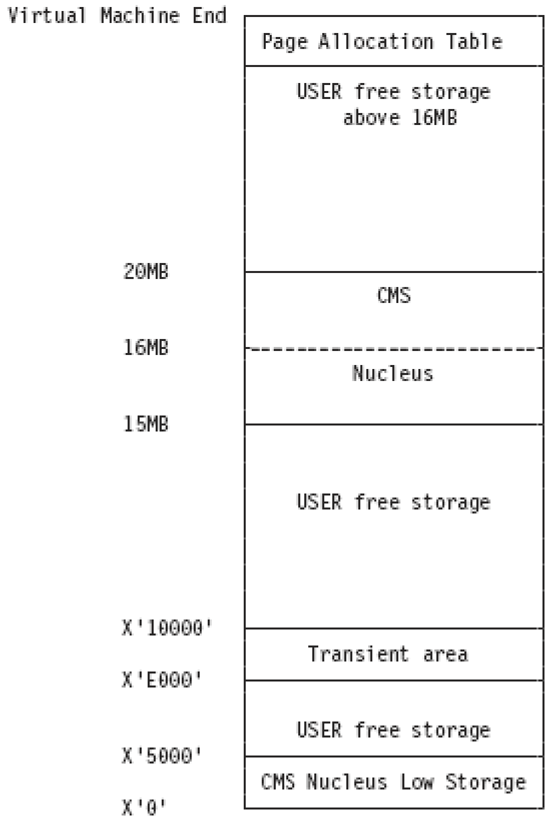


Figure 14. Storage Configuration for a Virtual Machine Greater than 20 MB

Obtaining Free Storage

Use CMSSTOR OBTAIN to allocate free storage. Although CMSSTOR OBTAIN provides a great deal of flexibility, the only thing you must specify when obtaining storage below 2 GB is the amount of storage you want. For example, to request 100 doublewords of storage, code:

```
CMSSTOR OBTAIN,DWORDS=100
```

or, to specify that register 2 contains the number of doublewords of storage you require, code:

```
CMSSTOR OBTAIN,DWORDS=(R2)
```

On return from CMSSTOR, register 0 contains the amount of storage allocated and register 1 contains the address of the storage allocated.

Where CMSSTOR Gets Storage

By default, CMSSTOR allocates storage based on the program's addressing mode:

1. For an AMODE 24 program, CMSSTOR allocates storage from the highest free storage location available below 16 MB.
2. For an AMODE 31 program, CMSSTOR allocates storage from the highest free storage location available. If an AMODE 31 program must obtain storage from below 16 MB (and the program is running in a virtual machine greater than 16 MB), it must explicitly specify so on the CMSSTOR macro. (You can use the LOC or ADDR parameters to specify an address below 16 MB.)

CMSSTOR Error Processing

It is also worth noting that, by default, your program abends if CMSSTOR encounters an error. If you do not want the program to abend, you can use the ERROR parameter to specify the address of an instruction where, following an error, execution continues. For example, to specify that the instruction at STORERR receive control, code:

```
CMSSTOR OBTAIN,DWORDS=(R2),ERROR=STORERR
```

The following section provides examples of how you can use the CMSSTOR parameters to tailor your free storage requests. For information on how to use CMSSTOR and SUBPOOL to manage free storage subpools, see "Creating Subpools" on page 57.

Where You Can Allocate Free Storage

For all of the following examples, assume that your program (a) has already stored in register 2 the number of doublewords of storage it requires and (b) has some error handling code defined at STORERR.

Example 1 — Specific Address

To allocate storage from the location specified in register 3, code:

```
CMSSTOR OBTAIN,DWORDS=(R2),ADDR=(R3),ERROR=STORERR
```

Note: You cannot allocate storage from an area in use or not contained within your virtual machine size.

Example 2 — Below 16 MB

To allocate storage from below 16 MB, code:

```
CMSSTOR OBTAIN,DWORDS=(R2),LOC=BELOW,ERROR=STORERR
```

Example 3 — Above 16 MB

To allocate storage from above 16 MB, code:

```
CMSSTOR OBTAIN,DWORDS=(R2),LOC=ABOVE,ERROR=STORERR
```

Note: Because an AMODE 24 program cannot address storage locations above 16 MB, they must not attempt to allocate storage from above 16 MB.

Example 4 — On a Page Boundary

By default, storage that CMS allocates is aligned on doubleword boundaries. To specify that the storage be aligned on a page boundary, code:

```
CMSSTOR OBTAIN,DWORDS=(R2),BNDRY=PAGE,ERROR=STORERR
```

Other Things You Can Specify on CMSSTOR

The following list describes the other parameters you can use with the CMSSTOR macros:

- TYPCALL — If the routine requesting storage is nucleus resident, specify TYPCALL=BRANCH on the CMSSTOR macro.
- MSG — If you do not want CMS to display a message when it encounters an error, specify MSG=NO on the CMSSTOR macro.
- MF — The MF parameter lets you specify various macro formats.

For more details on these parameters, see *z/VM: CMS Macros and Functions Reference*. In addition, the CMSSTOR macro includes a SUBPOOL parameter, which lets you allocate storage from a specific subpool. Subpools are described in the following sections.

Creating Subpools

By default, CMS organizes storage below 2 GB into two subpools named NUCLEUS and USER. You can obtain storage from these subpools or you can use the CMSSTOR and SUBPOOL macros to create your own subpools. The SUBPOOL and CMSSTOR macros allow you to:

- Manipulate subpools of storage as a single entity.
- Create storage subpools that are private to a program or global across SVC levels.
- Release an entire free storage subpool. (If a program does not explicitly delete a PRIVATE or SHARED subpool, CMS automatically deletes it when the program that created it terminates. This ensures that unused storage is released and maintains the integrity of free storage.)

To the application or system programmer, subpooling can be transparent. You do not have to use the SUBPOOL macro or the SUBPOOL parameter on the CMSSTOR macro. CMS can automatically satisfy free storage requests from the USER subpool.

Types of Subpools

There are three types of subpools you can create:

Table 15. Types of Subpools

Subpool	**Description**	**Exists until:**
PRIVATE	Satisfy only the subpool creator's requests for free storage. A program can do any manipulations it wants to a PRIVATE subpool without affecting other programs. Only one PRIVATE subpool with the same name can be active at a time; therefore, a single program can create several subpools with the same name. When CMS activates a new PRIVATE subpool with the same name as an existing subpool, it pushes the previous subpools back on the LIFO stack. When CMS deletes the current subpool, the previous one is popped from the stack and becomes active again.	• The program that created it explicitly deletes it, • SVC 202/CMSCALL termination, or • CMS performs abend recovery.
SHARED	Allow you to share free storage or data with other programs. Unlike PRIVATE subpools, SHARED subpools can be used by any program that the subpool creator calls, or that the called program calls, and so on. For example, assume PROGA creates a SUBPOOL named MINE. PROGA calls PROGB, PROGB calls PROGC, and PROGC calls PROGD. All of the programs in the calling chain (PROGA, PROGB, PROGC, and PROGD) can request free storage from MINE.	• Any program on the calling chain explicitly deletes it, • SVC 202/CMSCALL termination of the program that obtained the storage, or • CMS performs abend recovery.

Table 15. Types of Subpools (continued)

Subpool	Description	Exists until:
GLOBAL	Allow you to organize storage by its intended function (any program running in the virtual machine can use a global subpool) and to save data across program invocations (GLOBAL subpools still exist after SVC 202/CMSCALL termination). To retain GLOBAL subpools during abend recovery, specify SYSTEM=YES on the SUBPOOL macro. Using storage from GLOBAL subpools is useful when you create nucleus extensions that must retain data across invocations or abend processing.	• A program issues the SUBPOOL DELETE macro, or • CMS performs abend recovery if SYSTEM=NO is specified on the SUBPOOL macro.

Naming Subpools

There are no restrictions on the characters you can use for subpool names; however, the subpool names DMSxxxxx are reserved for system use and the names USER, USERG, and NUCLEUS are for reserved system subpools.

Subpool Examples

For all of the following examples, assume that the program has some error handling code defined at STORERR. For examples that use CMSSTOR, assume your program has already stored in register 2 the number of doublewords of storage it requires. For a description of the parameters not described in the following examples, see the *z/VM: CMS Macros and Functions Reference*.

Example 1 — Creating a Private Subpool

To create a private subpool named ALLMINE, code:

```
SUBPOOL CREATE,NAME='ALLMINE',ERROR=STORERR
```

You could also use the CMSSTOR macro to create a private subpool named ALLMINE and obtain storage from it:

```
CMSSTOR OBTAIN,DWORDS=(R2),SUBPOOL='ALLMINE',ERROR=STORERR
```

The difference between using CMSSTOR and SUBPOOL to create subpools is:

- CMSSTOR only creates subpools when it cannot find one with the name and type you specify.
- SUBPOOL creates a new subpool regardless of whether one with the same name exists. Because CMS keeps subpools on a stack, the new subpool is *pushed* on the stack on top of existing subpools with the same name.

Example 2 — Creating a Shared Subpool

To create a shared or global subpool, you must specify the name and type of the subpool. For example, to create a shared subpool named ALLMINE, code:

```
SUBPOOL CREATE,NAME='ALLMINE',TYPE=SHARED,ERROR=STORERR
```

Note: You cannot use CMSSTOR to create shared subpools.

Example 3 — Creating a Global Subpool

To create a GLOBAL subpool named ALLOURS, code:

```
SUBPOOL CREATE,NAME='ALLOURS',TYPE=GLOBAL,ERROR=STORERR
```

Or, to create a global subpool named ALLOURS and obtain storage from it:

```
CMSSTOR OBTAIN,DWORDS=(R2),SUBPOOL=('ALLOURS',GLOBAL)
```

Example 4 — Creating a Subpool in Nucleus Storage

By default, CMS allocates storage for your subpools from the USER subpool. To specify that CMS allocate storage for a private subpool named ALLMINE and that storage be in nucleus key, code:

```
SUBPOOL CREATE,NAME='ALLMINE',KEY=NUCLEUS,ERROR=STORERR
```

You cannot use the CMSSTOR macro to create named subpools in nucleus storage.

Example 5 — Saving Global Subpools Across Abends

During CMS abend processing, CMS, by default, releases all subpools and free storage that were not allocated from the NUCLEUS subpool. The SYSTEM parameter of the SUBPOOL macro lets you designate global subpools that you want to survive abend processing. For example, to specify that ALLOURS survive abend processing, code:

```
SUBPOOL CREATE,NAME='ALLOURS',TYPE=GLOBAL,SYSTEM=YES,ERROR=STORERR
```

If you do not specify TYPE=GLOBAL and SYSTEM=YES, CMS releases the storage when an abend occurs.

Releasing Free Storage

To release storage below 2 GB, you can (a) use CMSSTOR RELEASE to release specific blocks of storage allocated by CMSSTOR OBTAIN, (b) use SUBPOOL RELEASE to release storage associated with subpools created by CMSSTOR OBTAIN or SUBPOOL CREATE, or (c) use SUBPOOL DELETE to release the storage associated with subpools and delete the subpool off the chain. If you do not explicitly release storage, CMS does it automatically as shown in Table 16 on page 59.

Table 16. How CMS Releases Free Storage

Action	USER or USERG Subpool	NUCLEUS Subpool	Named Subpool	GLOBAL Subpool SYSTEM= YES	GLOBAL Subpool SYSTEM= NO
SVC 202 or CMSCALL Termination	retain	retain	delete	retain	retain
Abend Recovery	release	retain	delete	retain	delete

- SYSTEM=YES — as specified on the SUBPOOL macro, the GLOBAL subpool is to survive abend processing.
- SYSTEM=NO — as specified on the SUBPOOL macro, the GLOBAL subpool is not to survive abend processing.
- retain — the subpool is not affected by the action.
- release — the subpool is released (the storage associated with the subpool is returned although the subpool name remains on the queue of subpool names).
- delete — the subpool is deleted (the storage associated with the subpool is returned and the subpool name is deleted from the queue of subpool names).

Examples of Releasing Free Storage

Example 1 — Releasing a Specific Storage Block

For the first two examples, assume that your program has already (a) stored in register 2 the amount of storage to be released, (b) stored in register 3 the address of the storage, and (c) defined at location RELERR some code to handle errors that might occur during CMSSTOR processing. To release this storage, code:

```
CMSSTOR RELEASE,DWORDS=(R2),ADDR=(R3),ERROR=RELERR
```

Note: If CMSSTOR RELEASE is successful, it stores a 0 return code in register 15 and leaves in register 0 the amount of storage released.

Example 2 — Subpool Integrity Checking

To request that CMS make sure that the block of storage you release is owned by the subpool you specify (in this case, the global subpool named LARGE), code:

```
CMSSTOR RELEASE,DWORDS=(R2),ADDR=(R3),SUBPOOL=('LARGE',GLOBAL),
        ERROR=RELERR
```

If LARGE does not own the block of storage, CMS issues an error (RC=10) and does not release the storage.

Example 3 — Releasing or Deleting a Specific Private Subpool

To release the storage contained in the private subpool named ALLMINE, code:

```
SUBPOOL RELEASE,NAME='ALLMINE',ERROR=RELERR
```

To release the storage contained in the private subpool named ALLMINE and delete ALLMINE from the list of currently active subpools, code:

```
SUBPOOL DELETE,NAME='ALLMINE',ERROR=RELERR
```

Example 4 — Releasing or Deleting a Specific Shared or Global Subpool

To release or delete a shared or global subpool, specify the name and type of subpool. For example, to release the storage contained in a shared subpool named ALLMINE, code:

```
SUBPOOL RELEASE,NAME='ALLMINE',TYPE=SHARED,ERROR=RELERR
```

To release the storage contained in a global subpool named LARGE and delete ALLMINE from the list of currently active subpools, code:

```
SUBPOOL DELETE,NAME='LARGE',TYPE=GLOBAL,ERROR=RELERR
```

For more details on these parameters, see *z/VM: CMS Macros and Functions Reference*.

Determining How Much Free Storage Is Available

You may want to determine the amount of free storage available in your virtual machine, how this storage is allocated, or the current amount of free storage allocated to a subpool. The STORMAP and SUBPMAP commands provide you with this information for storage below 2 GB. These commands will not display storage or subpool information for virtual machine storage above 2 GB.

Using the STORMAP Command

The STORMAP command lets you determine the current utilization of free storage in your virtual machine. STORMAP provides you with information about the amount of allocated and unallocated free storage within your virtual machine as well as the size of the largest contiguous block of storage both above and below 16 MB, up to 2 GB.

See the *z/VM: CMS Commands and Utilities Reference* for details on the STORMAP command.

Example - Determining If a Piece of Contiguous Free Storage Exists

Before calling your application, you can execute the following REXX exec to determine if a 128K piece of contiguous free storage exists below 16 MB:

```
/* Determine if a 128K piece of contiguous storage exists  */
/* below 16MB.                                             */
/* Return "1" if it is available and "0" if it is not.     */

required = 128 * 1024              /* 128K in bytes        */

call 'STORMAP (STEM DATA.'         /* Get the STORMAP Data */

if result <> 0                     /* Clean call?          */
  then                             /* No, then call our    */
    call Error_Rtn                 /* Error handler...     */

if X2D(DATA.LGLT16MB) < required   /* Sufficient storage?  */
  then                             /* No, not enough...    */
    do
      say 'There is not a 128K piece of free storage below 16MB'
      flag = 0
    end
  else
    flag = 1

return flag
```

Note: This value is an approximation because the REXX processor uses storage during execution.

Example - Determining the Total Amount of Unallocated Storage on a Subpool

If your application uses a subpool for its storage requests, you can execute the following REXX exec to determine the total amount of unallocated storage on that specific subpool:

```
/* Return the amount of storage available for allocation   */
/* on the NUCLEUS subpool. The value is in hex.            */

call 'STORMAP (ALL STEM DATA.'     /* Get the STORMAP Data */

if result <> 0                     /* Clean call?          */
  then                             /* No, then call our    */
    call Error_Rtn                 /* Error handler...     */

return D2X(X2D(Data.TOTLUNAL) + X2D(Part_Sum('NUCLEUS')))
 Part_Sum:

procedure expose data.

parse arg spname .                           /* Get the subpool name  */
spname = left(spname,8)                      /* Make it 8 bytes long  */
total  = 0                                   /* Our summation var     */

do cnt = 1 to data.0                         /* Process the stem      */
  parse var data.cnt tstname 9 . . bytes . . attributes
  attributes = strip(attributes)

  if tstname ¬== spname
    then
      iterate                                /* Not our subpool       */

  if attributes = 'UNALLOC'                  /* Unallocated piece?    */
    then                                     /* Yes, and our subpool  */
      total = total + X2D(bytes)             /* Add in the bytes      */
end

return D2X(total)                            /* Total partial storage */
```

Note: Again, this is only an approximation because of the storage utilization by the REXX processor during execution.

Using the SUBPMAP Command

The SUBPMAP command lets you determine the current allocation of pages to storage subpools defined in your virtual machine. SUBPMAP provides you with information concerning the number of fully and partially allocated pages of storage for all subpools in your virtual machine or a specific list of subpools. The SUBPMAP command will not provide storage information for pages allocated above 2 GB.

See the *z/VM: CMS Commands and Utilities Reference* for details on the SUBPMAP command.

Example - Determining How Many Pages of Storage are Allocated to a Subpool

The following REXX exec determines if a particular subpool exists, and if the subpool does exist, this exec displays the number of pages of storage allocated to it:

```
/* Determine if a subpool exists and how many pages of storage     */
/* are allocated to it.                                            */

parse upper arg spname .            /* Get the name of the subpool */
spname = left(spname,8)             /* Make it 8 bytes long        */

call 'SUBPMAP' spname '(STEM DATA.' /* Get our data from SUBPMAP   */

select
  when result = 10                  /* Was the subpool found?      */
    then                            /* No, tell the caller         */
      say spname 'does not exist'
  when result <> 0                  /* Some other error?           */
    then                            /* Yes, call the error handler */
      call Error_Rtn
  otherwise                         /* Add partial and full pages  */
    do
      parse var data.1 . . . part full .
      total = part + full
      say spname 'has' total 'pages allocated to it'
    End
 End

return 0
```

Debugging Storage Problems

Your application may require debugging because of storage problems. You can use the STORMAP, SUBPMAP, and STDEBUG commands to help you determine the cause of the storage problem, for storage obtained and released below 2 GB.

The STORMAP and SUBPMAP commands provide you with information about the storage in your virtual machine and the pages of storage for subpools in your virtual machine. The information may include the name of the subpool that owns the storage, the address of the subpool descriptor block, the start address of the piece of storage being mapped, the end address of the piece of storage being mapped, the number of bytes being mapped, the number of pages being mapped, the storage protection key of the page, and the storage attributes. This information is displayed on your console or written to a file.

The STDEBUG command traces the obtain and release requests made by your application. The trace information includes the number of bytes obtained or released, the address of storage obtained or released, the name of the subpool that owns the storage, and the address of the caller to storage management. This information is displayed on your console or written to a unit record device.

See the *z/VM: CMS Commands and Utilities Reference* for details on the STORMAP, SUBPMAP, and STDEBUG commands.

Using the STORMAP Command

The STORMAP command lets you control the specific storage address range or subpool to be mapped, when you want the storage mapped, and the type of storage to be mapped, such as allocated storage, unallocated storage, partial pages of storage, and full pages of storage. Therefore, you can generate the data only needed to perform the debugging operation.

See the *z/VM: CMS Commands and Utilities Reference* for details on the STORMAP command.

Example - Using STORMAP with the EXTSET Option

Suppose you want to generate storage information whenever you execute a specific BALR instruction. You can use the STORMAP command with the EXTSET option and the CP TRACE command, in addition to other commands, to produce information about the amount of free storage available each time the BALR instruction is executed.

The following example shows you how this debugging works:

```
stormap (extset 250
Ready;
loadmod sttest
Ready;
progmap
Name        Entry       Origin      Bytes        Attributes
STTEST      013E6DB0    013E6DB0    00000250    Amode 31  Reloc
Ready;
d i13e6db0.10
R013E6DB0  LR    18CF         LA    41500005     L     58F0C244     E6
R013E6DBA  BALR  05EF         BCT   4650C006
Ready;
trace inst pswa 13e6dba run cmd ext 250
Ready;
start
DMSLIO740I Execution begins...
 -> 013E6DBA  BALR  05EF     -> 013E6FBA    CC 1
                                   Storage Map
                                  ------- ---
 VMSIZE       NUCALPHA     NUCSIGMA     NUCOMEGA     NUCPHI       NUCCHI
01400000     00F00000     00F52A00     01300000     01000000    012CF5C8
                           Unallocated Free Storage Queue
                          ----------- ---- ------- -----
       <16MB                         >16MB
Total       Largest        Total       Largest      Total Unallocated
00D2B000    00D1E000       003E1000    003DB000     0110C000
 -> 013E6DBA  BALR  05EF     -> 013E6FBA    CC 2
                                   Storage Map
                                  ------- ---
 VMSIZE       NUCALPHA     NUCSIGMA     NUCOMEGA     NUCPHI       NUCCHI
01400000     00F00000     00F52A00     01300000     01000000    012CF5C8
```

```
                            Unallocated Free Storage Queue
                            ----------- ---- ------- -----
      <16MB                       >16MB
Total       Largest        Total       Largest      Total Unallocated
00D2B000    00D1E000       003DF000    003DB000     0110A000
 -> 013E6DBA  BALR  05EF     -> 013E6FBA    CC 2
                                    Storage Map
                                    ------- ---
VMSIZE       NUCALPHA     NUCSIGMA     NUCOMEGA     NUCPHI       NUCCHI
01400000     00F00000     00F52A00     01300000     01000000 012CF5C8
                            Unallocated Free Storage Queue
                            ----------- ---- ------- -----
      <16MB                       >16MB
Total       Largest        Total       Largest      Total Unallocated
00D2B000    00D1E000       003DD000    003DB000     01108000
 -> 013E6DBA  BALR  05EF     -> 013E6FBA    CC 2
                                    Storage Map
                                    ------- ---
 VMSIZE       NUCALPHA     NUCSIGMA     NUCOMEGA     NUCPHI       NUCCHI
01400000     00F00000     00F52A00     01300000    01000000 012CF5C8
                            Unallocated Free Storage Queue
                            ----------- ---- ------- -----
       <16MB                       >16MB
Total       Largest        Total       Largest      Total Unallocated
00D2B000    00D1E000       003DB000    003D9000     01106000
 -> 013E6DBA  BALR  05EF     -> 013E6FBA    CC 2
                                    Storage Map
                                    ------- ---
 VMSIZE       NUCALPHA     NUCSIGMA     NUCOMEGA     NUCPHI       NUCCHI
01400000     00F00000     00F52A00     01300000     01000000 012CF5C8
                            Unallocated Free Storage Queue
                            ----------- ---- ------- -----
       <16MB                       >16MB
Total       Largest        Total       Largest      Total Unallocated
00D2B000    00D1E000       003D9000    003D7000     01104000 Ready;
```

The following steps describe the procedure illustrated in the previous example:

- The STORMAP command generates summary information whenever an external interrupt X'250' is reflected to the virtual machine. This external interrupt corresponds to CMD EXT 250 set in the TRACE instruction issued later.
- The LOADMOD command loads your STTEST program.
- The PROGMAP command determines where in storage STTEST is loaded.
- After you know the load point, 013E6DB0, the CP DISPLAY I command locates the BALR instruction. This is the instruction you want to trace.
- Then, the CP TRACE command sets a trace for the BALR instruction. The RUN option causes execution to continue when the BALR is executed. The CMD EXT X'250' generates an external interruptX'250' when the BALR is executed. This external interrupt corresponds to the EXTSET 250 option set on the STORMAP command issued earlier.
- The START command starts execution of STTEST. The result is that each time the BALR is executed, an external interrupt X'250' is generated which in turn displays the storage information.

The data shows that the total amount of unallocated storage above 16 MB is reduced by two pages on each BALR call. You now have to determine if this loss of two pages is a normal condition on that call.

After you know the meaning of the fields in the summary report, you can reduce the amount of data displayed by using the NOHEAD option on the STORMAP command. For example:

```
stormap (nohead extset 255
Ready;
d i13e6910.10
R013E6910  LR    18CF         LA    41500005     L     58F0C244     E6
R013E691A  BALR  05EF         BCT   4650C006
Ready;
trace inst pswa 13e691a run cmd ext 255
Ready;
start
DMSLIO740I Execution begins...
 -> 013E691A  BALR  05EF     -> 013E6B1A    CC 1
01400000     00E00000     00FEB158     01000000
00D2B000    00D1E000       003CD000    003CB000     010F8000
 -> 013E691A  BALR  05EF     -> 013E6B1A    CC 2
01400000     00E00000     00FEB158     01000000
00D2B000    00D1E000       003CB000    003C9000     010F6000
 -> 013E691A  BALR  05EF     -> 013E6B1A    CC 2
01400000     00E00000     00FEB158     01000000
00D2B000    00D1E000       003C9000    003C7000     010F4000
 -> 013E691A  BALR  05EF     -> 013E6B1A    CC 2
01400000     00E00000     00FEB158     01000000
00D2B000    00D1E000       003C7000    003C5000     010F2000
 -> 013E691A  BALR  05EF     -> 013E6B1A    CC 2
01400000     00E00000     00FEB158     01000000
00D2B000    00D1E000       003C5000    003C3000     010F0000
Ready;
```

The third token of the second line of data after each BALR is reduced by two pages. From our previous example, you know this field is for unallocated storage above 16 MB.

Note: Remember, you can also use the FILE APPEND option to write the data to the

STORMAP DATA A file rather than to the console.

Example - Using STORMAP to Determine if Storage Fragmentation Exists

Suppose there is a large amount of free storage available on your virtual machine, but there is not one piece of storage large enough to satisfy an obtain request. This situation may occur because of storage fragmentation. Storage fragmentation generally occurs when the pattern of storage obtain and release requests leaves many small pieces of unallocated storage interleaved with allocated pieces. You can use the STORMAP command to determine if storage fragmentation exists.

In this example, the following STORMAP command reveals a storage fragmentation condition in the TSTPOOL subpool:

```
stormap (subpool TSTPOOL
                                  Storage Map
                                  ------- ---


Address Range: 00000000 - 00FFFFFF

Subpool   Start     End       Bytes     Pages   Key  Attributes
TSTPOOL   00D72000  00D724BF  000004C0      p   E0   ALLOC   GLOBAL SYSTEM
TSTPOOL   00D724C0  00D724FF  00000040      p   E0   UNALLOC
TSTPOOL   00D72500  00D72FFF  00000B00      p   E0   ALLOC   GLOBAL SYSTEM
TSTPOOL   00D73000  00D734BF  000004C0      p   E0   ALLOC   GLOBAL SYSTEM
TSTPOOL   00D734C0  00D734FF  00000040      p   E0   UNALLOC
TSTPOOL   00D73500  00D73FFF  00000B00      p   E0   ALLOC   GLOBAL SYSTEM
TSTPOOL   00D74000  00D744BF  000004C0      p   E0   ALLOC   GLOBAL SYSTEM
TSTPOOL   00D744C0  00D744FF  00000040      p   E0   UNALLOC
TSTPOOL   00D74500  00D74FFF  00000B00      p   E0   ALLOC   GLOBAL SYSTEM
TSTPOOL   00D75000  00D754BF  000004C0      p   E0   ALLOC   GLOBAL SYSTEM
TSTPOOL   00D754C0  00D754FF  00000040      p   E0   UNALLOC
TSTPOOL   00D75500  00D75FFF  00000B00      p   E0   ALLOC   GLOBAL SYSTEM
TSTPOOL   00D76000  00D764BF  000004C0      p   E0   ALLOC   GLOBAL SYSTEM
TSTPOOL   00D764C0  00D764FF  00000040      p   E0   UNALLOC
TSTPOOL   00D76500  00D76FFF  00000B00      p   E0   ALLOC   GLOBAL SYSTEM
TSTPOOL   00D77000  00D774BF  000004C0      p   E0   ALLOC   GLOBAL SYSTEM
TSTPOOL   00D774C0  00D774FF  00000040      p   E0   UNALLOC
TSTPOOL   00D77500  00D77FFF  00000B00      p   E0   ALLOC   GLOBAL SYSTEM
```

Each page on the TSTPOOL subpool has a X'40' byte piece of unallocated storage. Two different sizes of storage are requested on TSTPOOL—one piece is X'4C0' bytes and the other piece is X'B00' bytes. On a page of X'1000' bytes, this leaves X'40' bytes remaining. You may not be able to change this situation; however, you can examine the application and determine if any pieces of free storage obtained on some other subpool would fit into these X'40' byte pieces of unallocated storage. In some situations, it is desirable to split storage requests onto two different subpools. See "Storage Fragmentation" on page 69 for an example of two different subpools.

Using the SUBPMAP Command

During application debugging, you can use the SUBPMAP command to determine if specific subpools have allocated storage at a given point in time. You can also use SUBPMAP as a check at end of command to determine if any storage remains on subpools that were supposed to be cleaned up during application termination. When you use the EXTSET option, the status of storage allocation on subpools are displayed during CP TRACE operations, provided the application is enabled for external interrupts.

See the *z/VM: CMS Commands and Utilities Reference* for details on the SUBPMAP command.

Example - Using SUBPMAP

The following program, when loaded as a nucleus extension with the ENDCMD option, displays a single line at end of command processing. This line contains allocated storage information about the USER subpool when each command or program terminates.

```
SUBEND   CSECT
         USING SUBEND,R12
         LR    R12,R15
         CMSCALL PLIST=PLIST,EPLIST=EPLIST
         BR    R14
         SPACE 1
PLIST    DC    CL8'SUBPMAP'
         DC    XL4'FFFFFFFF'
         SPACE 1
EPLIST   DC    A(PLIST)
         DC    A(SUSER)
         DC    A(EUSER)
         DC    A(0)
         SPACE 1
SUSER    DC    C'USER (NOHEAD'
EUSER    DS    X
         LTORG
         REGEQU
         END
```

The output from SUBEND is similar to the following:

```
nucxload subend (endcmd
USER      E0    013FFC2C         0         1  GLOBAL
Ready;
testprog
USER      E0    013FFC2C         2         3  GLOBAL
Ready;
```

According to this output, after executing TESTPROG, the address of the USER subpool descriptor block is 013FFC2C, there are 2 full pages of allocated storage and 3 partial pages of allocated storage on the USER subpool, and the USER subpool is a GLOBAL subpool.

Using the STDEBUG Command

You can use the STDEBUG command to trace calls to storage management to determine what caused a storage problem, such as an out of storage condition, storage fragmentation, or storage overlays. With the trace enabled, you can trap CMSSTOR, DMSFREE, DMSFRET, FREEMAIN, and GETMAIN requests providing you with the following information:

- Amount of storage obtained or released
- Address of the storage obtained or released
- Address of the caller to storage management

- Name of the subpool the storage is allocated on.

See the *z/VM: CMS Commands and Utilities Reference* for details on the STDEBUG command.

The STDEBUG command also lets you control the caller address range to be traced, the storage address range (the area where storage is being obtained or released from) to be traced, and the list of subpools to be traced. Therefore, you can generate the data only needed to perform the debugging operation.

For example, when a program in a known storage location requires debugging, it is not necessary to trace all the calls to storage management within the CMS nucleus. You can limit the trace to only the calls made within a specific storage address range.

Another example is when a repeatable error condition, such as a program exception, occurs when you run your application after an abend or IPL. This can happen because data (such as a branch address) is retrieved from a storage location that is no longer allocated. By limiting the trace to only the address range of the storage that is supposed to contain valid data, you can easily identify who obtained and released the storage.

Out of Storage Conditions

Out of storage conditions occur when either a request has been made for storage and no available block of unallocated storage is large enough to satisfy the request, or when a request has been made for storage at a specific address and all or a portion of it is already allocated. An out of storage condition may occur because of one of the following:

- Programming error causing storage to never be released
- Storage fragmentation
- Insufficient virtual machine size to satisfy the number of free storage requests.

Determining Causes for Storage Not Being Released

You can use the following methods to determine the cause for storage not being released:

- Use the MSG option on the STDEBUG command. The MSG option causes storage management error messages to be issued even though the caller to storage management specified MSG=NO.
- Use the OBTAIN and RELEASE options on the STDEBUG command. The OBTAIN and RELEASE options enable storage tracing for obtain and release requests. When the application has completed and the generated data is displayed, find the corresponding RELEASE for each OBTAIN. If an OBTAIN is found that does not have a matching RELEASE, it is possible that this storage has never been cleaned up, therefore, causing your out of storage condition.

Example - Using the MSG Option on STDEBUG

Suppose your application contains the following subroutine to release free storage:

```
**************************************************************
* Input:
*
*   R0  = Number of bytes to release
*   R2  = Address of storage to be released
*   R14 = Return address
**************************************************************
         SPACE 1
RELSTORE EQU   *
         CMSSTOR RELEASE,BYTES=(0),ADDR=(2),MSG=NO,ERROR=*
         BR    R14
```

If an error occurs on the CMSSTOR RELEASE, a message is not issued because MSG=NO is specified. In addition, this routine is not checking the return code for an error.

Now, suppose an error existed in the calling program so an invalid address was passed to this subroutine on each invocation. This may be caused by improper addressability to a control block. The result is that the actual pieces of free storage are never released and an out of storage condition may occur.

The following example shows how you can use the MSG option on the STDEBUG command to detect this situation:

```
Ready;
msgtest
Ready;
stdebug (msg
Ready;
msgtest
DMSFRR161E Invalid free storage release call from 013E803E, error code 6
DMSFRR161E Invalid free storage release call from 013E803E, error code 6
DMSFRR161E Invalid free storage release call from 013E803E, error code 6
DMSFRR161E Invalid free storage release call from 013E803E, error code 6
DMSFRR161E Invalid free storage release call from 013E803E, error code 6
Ready;
```

Issuing the STDEBUG command with the MSG option before executing your application, MSGTEST, reveals the error in CMSSTOR RELEASE that would normally be suppressed.

Example - Using the OBTAIN and RELEASE Options on STDEBUG and Pairing Obtain and Release Requests

Your application may encounter an out of storage condition because storage was obtained but never released. This can occur in cooperative applications where one application obtains storage and it is the responsibility for another application to release it.

Consider the following execution of TEST1 with a storage trace enabled for all obtain and release requests:

```
Ready;
stdebug (ob re punch d
Ready;
test1
Ready;
stdebug (end
Ready;
close pun to * rdr name pun log
RDR FILE 0108 SENT FROM FRODO    PUN WAS 0108 RECS 0047 CPY  001 A NOHOLD NOKEEP
Ready;
```

The resulting punch file appears as follows:

```
15:09:03 OBTAINED BYTES=00000130 ADDR=00CC8518 SUBPL=NUCLEUS  CALLER=00E1BB06
15:09:03 RELEASED BYTES=00000130 ADDR=00CC8518 SUBPL=NUCLEUS  CALLER=00E0E792
15:09:03 OBTAINED BYTES=000000E0 ADDR=00D17878 SUBPL=USER     CALLER=00E35CB6
15:09:03 RELEASED BYTES=000000E0 ADDR=00D17878 SUBPL=USER     CALLER=00E35F72
15:09:08 OBTAINED BYTES=00000130 ADDR=00CC8518 SUBPL=NUCLEUS  CALLER=00E1BB06
15:09:08 RELEASED BYTES=00000130 ADDR=00CC8518 SUBPL=NUCLEUS  CALLER=00E0E792
15:09:08 OBTAINED BYTES=00000800 ADDR=013EA6C8 SUBPL=DMSBLOKN CALLER=00E2B404
15:09:08 OBTAINED BYTES=00000050 ADDR=013EA678 SUBPL=DMSBLOKN CALLER=00E45182
15:09:08 OBTAINED BYTES=000000A8 ADDR=013E8000 SUBPL=DMSUSRM  CALLER=00E45644
15:09:08 RELEASED BYTES=00000050 ADDR=013EA678 SUBPL=DMSBLOKN CALLER=00E4638C
15:09:08 RELEASED BYTES=00000800 ADDR=013EA6C8 SUBPL=DMSBLOKN CALLER=00E29404
15:09:08 OBTAINED BYTES=00000020 ADDR=013E6000 SUBPL=DMSBLOKU CALLER=00E4675A
15:09:08 OBTAINED BYTES=00000010 ADDR=013E5E00 SUBPL=USER     CALLER=013E803E
15:09:08 OBTAINED BYTES=00000010 ADDR=013E5DF0 SUBPL=USER     CALLER=013E803E
15:09:08 OBTAINED BYTES=00000010 ADDR=013E5DE0 SUBPL=USER     CALLER=013E803E
15:09:08 OBTAINED BYTES=00000010 ADDR=013E5DD0 SUBPL=USER     CALLER=013E803E
15:09:08 OBTAINED BYTES=00000010 ADDR=013E5DC0 SUBPL=USER     CALLER=013E803E
15:09:08 RELEASED BYTES=00000010 ADDR=013E5E00 SUBPL=USER     CALLER=013E808A
15:09:08 RELEASED BYTES=00000010 ADDR=013E5DF0 SUBPL=USER     CALLER=013E808A
15:09:08 RELEASED BYTES=00000010 ADDR=013E5DE0 SUBPL=USER     CALLER=013E808A
15:09:08 RELEASED BYTES=00000010 ADDR=013E5DD0 SUBPL=USER     CALLER=013E808A
15:09:08 RELEASED BYTES=000000A8 ADDR=013E8000 SUBPL=DMSUSRM  CALLER=00E46BFC
15:09:08 RELEASED BYTES=00000020 ADDR=013E6000 SUBPL=DMSBLOKU CALLER=00E46CD0
15:09:12 OBTAINED BYTES=00000130 ADDR=00CC8518 SUBPL=NUCLEUS  CALLER=00E1BB06
15:09:12 RELEASED BYTES=00000130 ADDR=00CC8518 SUBPL=NUCLEUS  CALLER=00E0E792
```

If you pair this data by the address of storage, the size, and the time stamp, you can identify the following paired and unpaired storage requests:

```
15:09:03 OBTAINED BYTES=00000130 ADDR=00CC8518 SUBPL=NUCLEUS  CALLER=00E1BB06
15:09:03 RELEASED BYTES=00000130 ADDR=00CC8518 SUBPL=NUCLEUS  CALLER=00E0E792
15:09:03 OBTAINED BYTES=000000E0 ADDR=00D17878 SUBPL=USER     CALLER=00E35CB6
15:09:03 RELEASED BYTES=000000E0 ADDR=00D17878 SUBPL=USER     CALLER=00E35F72
15:09:08 OBTAINED BYTES=00000130 ADDR=00CC8518 SUBPL=NUCLEUS  CALLER=00E1BB06
15:09:08 RELEASED BYTES=00000130 ADDR=00CC8518 SUBPL=NUCLEUS  CALLER=00E0E792
15:09:08 OBTAINED BYTES=00000800 ADDR=013EA6C8 SUBPL=DMSBLOKN CALLER=00E2B404
15:09:08 RELEASED BYTES=00000800 ADDR=013EA6C8 SUBPL=DMSBLOKN CALLER=00E29404
15:09:08 OBTAINED BYTES=00000050 ADDR=013EA678 SUBPL=DMSBLOKN CALLER=00E45182
15:09:08 RELEASED BYTES=00000050 ADDR=013EA678 SUBPL=DMSBLOKN CALLER=00E4638C
15:09:08 OBTAINED BYTES=000000A8 ADDR=013E8000 SUBPL=DMSUSRM  CALLER=00E45644
15:09:08 RELEASED BYTES=000000A8 ADDR=013E8000 SUBPL=DMSUSRM  CALLER=00E46BFC
15:09:08 OBTAINED BYTES=00000020 ADDR=013E6000 SUBPL=DMSBLOKU CALLER=00E4675A
15:09:08 RELEASED BYTES=00000020 ADDR=013E6000 SUBPL=DMSBLOKU CALLER=00E46CD0
15:09:08 OBTAINED BYTES=00000010 ADDR=013E5E00 SUBPL=USER     CALLER=013E803E
15:09:08 RELEASED BYTES=00000010 ADDR=013E5E00 SUBPL=USER     CALLER=013E808A
15:09:08 OBTAINED BYTES=00000010 ADDR=013E5DF0 SUBPL=USER     CALLER=013E803E
15:09:08 RELEASED BYTES=00000010 ADDR=013E5DF0 SUBPL=USER     CALLER=013E808A
15:09:08 OBTAINED BYTES=00000010 ADDR=013E5DE0 SUBPL=USER     CALLER=013E803E
15:09:08 RELEASED BYTES=00000010 ADDR=013E5DE0 SUBPL=USER     CALLER=013E808A
15:09:08 OBTAINED BYTES=00000010 ADDR=013E5DD0 SUBPL=USER     CALLER=013E803E
15:09:08 RELEASED BYTES=00000010 ADDR=013E5DD0 SUBPL=USER     CALLER=013E808A
15:09:12 OBTAINED BYTES=00000130 ADDR=00CC8518 SUBPL=NUCLEUS  CALLER=00E1BB06
15:09:12 RELEASED BYTES=00000130 ADDR=00CC8518 SUBPL=NUCLEUS  CALLER=00E0E792

15:09:08 OBTAINED BYTES=00000010 ADDR=013E5DC0 SUBPL=USER     CALLER=013E803E
```

One X'10' byte piece of storage was not released. Each invocation of TEST1 results in an additional X'10' byte piece of storage that is not released. After hours of execution, invoking this application, an out of storage condition may occur.

Example - Using the OBTAIN and RELEASE Options on STDEBUG and Pairing Obtain and Release Requests

An out of storage condition can also occur when only part of the storage obtained is actually released. In this situation, the residual amount of storage not being released can accumulate until storage becomes exhausted.

To detect this problem, you can use the STDEBUG command to enable tracing for obtain and release requests. After you execute your program, suppose the following data is displayed:

```
07:53:15  * MSG FROM FRODO   : OBTAINED 00000050 00CA3230 NUCLEUS  013E859C
07:53:15  * MSG FROM FRODO   : RELEASED 0000004D 00CA3230 NUCLEUS  013E83D2
07:53:30  * MSG FROM FRODO   : OBTAINED 00000130 00CA3150 NUCLEUS  013E8B06
07:53:30  * MSG FROM FRODO   : RELEASED 00000128 00CA3150 NUCLEUS  013E8792
```

Then, you can pair the obtain and release requests to find the problem area. Two pieces of storage have been obtained and released. The size on the RELEASE for both pairs does not match the size on the OBTAIN. For the pair at address X'00CA3230', this is not important. It is not important because the size on the RELEASE, X'4D', when rounded up to a multiple of eight, is X'50', which is the size on the OBTAIN. (CMS storage management always rounds sizes that are not a multiple of eight to the next doubleword value.)

There is a potential problem for the pair at address X'00CA3150'. The size of the storage obtained was X'130' bytes; however, the size of the storage released was X'128' bytes. This leaves X'8' bytes of storage not accounted for. If this application is used repetitively, an out of storage condition can occur as the unclaimed storage accumulates.

Storage Fragmentation

Storage fragmentation generally occurs when the pattern of storage obtain and release requests leaves many small pieces of unallocated storage interleaved with allocated pieces. A large amount of free storage may be available in the virtual machine; however, no single piece is large enough to satisfy a large request. This can cause an out of storage condition.

You can use the STDEBUG command to detect when storage fragmentation is occurring.

Example - Using STDEBUG to Detect Storage Fragmentation

Suppose your application, SUBTEST, uses the subpool, MYSUB, for all its storage requests. The following data is generated after enabling the tracing of the obtain and release requests on the MYSUB subpool and invoking SUBTEST:

```
stdebug MYSUB (ob re
Ready;
subtest
09:05:08  * MSG FROM FRODO   : OBTAINED 00000800 00DD6000 MYSUB    00DDC03E
09:05:08  * MSG FROM FRODO   : OBTAINED 00000408 00DD6BF8 MYSUB    00DDC072
09:05:08  * MSG FROM FRODO   : OBTAINED 00000800 00DD5000 MYSUB    00DDC03E
09:05:08  * MSG FROM FRODO   : OBTAINED 00000408 00DD5BF8 MYSUB    00DDC072
09:05:08  * MSG FROM FRODO   : OBTAINED 00000800 00DD4000 MYSUB    00DDC03E
09:05:08  * MSG FROM FRODO   : OBTAINED 00000408 00DD4BF8 MYSUB    00DDC072
09:05:08  * MSG FROM FRODO   : OBTAINED 00000800 00DD3000 MYSUB    00DDC03E
09:05:08  * MSG FROM FRODO   : OBTAINED 00000408 00DD3BF8 MYSUB    00DDC072
09:05:08  * MSG FROM FRODO   : OBTAINED 00000800 00DD1000 MYSUB    00DDC03E
09:05:08  * MSG FROM FRODO   : OBTAINED 00000408 00DD1BF8 MYSUB    00DDC072
09:05:08  * MSG FROM FRODO   : OBTAINED 00000800 00DCA000 MYSUB    00DDC03E
09:05:08  * MSG FROM FRODO   : OBTAINED 00000408 00DCABF8 MYSUB    00DDC072
09:05:08  * MSG FROM FRODO   : OBTAINED 00000800 00DC9000 MYSUB    00DDC03E
09:05:08  * MSG FROM FRODO   : OBTAINED 00000408 00DC9BF8 MYSUB    00DDC072
09:05:08  * MSG FROM FRODO   : OBTAINED 00000800 00DC8000 MYSUB    00DDC03E
09:05:08  * MSG FROM FRODO   : OBTAINED 00000408 00DC8BF8 MYSUB    00DDC072
09:05:08  * MSG FROM FRODO   : RELEASED 00000408 00DD6BF8 MYSUB    00DDC0BE
09:05:08  * MSG FROM FRODO   : RELEASED 00000408 00DD5BF8 MYSUB    00DDC0BE
09:05:08  * MSG FROM FRODO   : RELEASED 00000408 00DD4BF8 MYSUB    00DDC0BE
09:05:08  * MSG FROM FRODO   : RELEASED 00000408 00DD3BF8 MYSUB    00DDC0BE
09:05:08  * MSG FROM FRODO   : RELEASED 00000408 00DD1BF8 MYSUB    00DDC0BE
09:05:08  * MSG FROM FRODO   : RELEASED 00000408 00DCABF8 MYSUB    00DDC0BE
09:05:08  * MSG FROM FRODO   : RELEASED 00000408 00DC9BF8 MYSUB    00DDC0BE
09:05:08  * MSG FROM FRODO   : RELEASED 00000408 00DC8BF8 MYSUB    00DDC0BE
Ready;
subpmap MYSUB
                                   Subpool Map
                                   ------- ---

Subpool   Key   SUBBK         Full      Part  Attributes
MYSUB     E0    00DFFE5C         0         8  GLOBAL
Ready;
```

The data generated by the STDEBUG command reveals the following pattern occurring:

- X'800' byte piece of storage and a X'408' byte piece of storage are obtained on each page.
- X'3F8' bytes of storage are left on each page.

When the application completes execution, all the X'408' bytes of storage are released and the X'800' byte pieces persist—they are used later. This results in 8 pages of storage remaining on the MYSUB subpool, each containing one X'800' byte piece of storage. The SUBPMAP verifies these results.

Now, rework the application, SUBTEST, so it uses two subpools for its storage requests. One subpool, MYSUB1, can be used for the X'800' byte pieces of storage. The second subpool, MYSUB2, can be used for the X'408' bytes pieces of storage.

The following data is generated after enabling storing tracing for obtain and release requests on the MYSUB1 and MYSUB2 subpools and invoking SUBTEST:

```
stdebug MYSUB1 MYSUB2 (ob re
Ready;
subtest
08:58:54  * MSG FROM FRODO   : OBTAINED 00000800 00DC6000 MYSUB1   00DDA03E
08:58:54  * MSG FROM FRODO   : OBTAINED 00000408 00DC5000 MYSUB2   00DDA072
08:58:54  * MSG FROM FRODO   : OBTAINED 00000800 00DC6800 MYSUB1   00DDA03E
08:58:54  * MSG FROM FRODO   : OBTAINED 00000408 00DC5BF8 MYSUB2   00DDA072
08:58:54  * MSG FROM FRODO   : OBTAINED 00000800 00DC4000 MYSUB1   00DDA03E
08:58:54  * MSG FROM FRODO   : OBTAINED 00000408 00DC57F0 MYSUB2   00DDA072
08:58:54  * MSG FROM FRODO   : OBTAINED 00000800 00DC4800 MYSUB1   00DDA03E
08:58:54  * MSG FROM FRODO   : OBTAINED 00000408 00DC3000 MYSUB2   00DDA072
08:58:54  * MSG FROM FRODO   : OBTAINED 00000800 00D38000 MYSUB1   00DDA03E
08:58:54  * MSG FROM FRODO   : OBTAINED 00000408 00DC3BF8 MYSUB2   00DDA072
08:58:54  * MSG FROM FRODO   : OBTAINED 00000800 00D38800 MYSUB1   00DDA03E
08:58:54  * MSG FROM FRODO   : OBTAINED 00000408 00DC37F0 MYSUB2   00DDA072
08:58:54  * MSG FROM FRODO   : OBTAINED 00000800 00D37000 MYSUB1   00DDA03E
08:58:54  * MSG FROM FRODO   : OBTAINED 00000408 00D1F000 MYSUB2   00DDA072
08:58:54  * MSG FROM FRODO   : OBTAINED 00000800 00D37800 MYSUB1   00DDA03E
08:58:54  * MSG FROM FRODO   : OBTAINED 00000408 00D1FBF8 MYSUB2   00DDA072
08:58:54  * MSG FROM FRODO   : RELEASED 00000408 00DC5000 MYSUB2   00DDA0BE
08:58:54  * MSG FROM FRODO   : RELEASED 00000408 00DC5BF8 MYSUB2   00DDA0BE
08:58:54  * MSG FROM FRODO   : RELEASED 00000408 00DC57F0 MYSUB2   00DDA0BE
08:58:54  * MSG FROM FRODO   : RELEASED 00000408 00DC3000 MYSUB2   00DDA0BE
08:58:54  * MSG FROM FRODO   : RELEASED 00000408 00DC3BF8 MYSUB2   00DDA0BE
08:58:54  * MSG FROM FRODO   : RELEASED 00000408 00DC37F0 MYSUB2   00DDA0BE
08:58:54  * MSG FROM FRODO   : RELEASED 00000408 00D1F000 MYSUB2   00DDA0BE
08:58:54  * MSG FROM FRODO   : RELEASED 00000408 00D1FBF8 MYSUB2   00DDA0BE
Ready;
subpmap MYSUB1 MYSUB2
                                  Subpool Map
                                  ------- ---

Subpool   Key   SUBBK         Full      Part  Attributes
MYSUB1    E0    00DFFE84         4         0  GLOBAL
MYSUB2    E0    00DFFEAC         0         0  GLOBAL
Ready;
```

This data reveals:

- Two X'800' byte pieces of storage are obtained for each page on the MYSUB1 subpool.
- Three X'408' byte pieces of storage are obtained for each page on the MYSUB2 subpool.

When the application completes executing, all the X'408' byte pieces of storage are released. A total number of four pages are left. Again, the SUBPMAP command verifies these results.

By using two subpools for storage requests, you reduced the amount of persistent storage used by this application by 50%. You were able to detect this fragmentation problem using the STDEBUG command.

Insufficient Virtual Machine Storage Size

You can also encounter an out of storage condition if your virtual machine storage size is not large enough to satisfy the number of free storage requests. If this occurs, you can increase your virtual machine storage size using the CP DEFINE STORAGE command. See the *z/VM: CP Commands and Utilities Reference* for details on the DEFINE STORAGE command.

Storage Overlays

Suppose you received the following messages after executing your application, OVTEST1:

```
Ready;
ovtest1
DMSFRO163E User key pointers have been destroyed (internal error code 85)
DMSFRM165I Chain header at address: 013FF610, Page address: 013E9000
DMSFRM817I Subpool name: ZORCH Subbk address: 013FFD44
```

To determine what caused this problem, you can trap each call for storage on the ZORCH subpool to determine where the application is storing into unallocated free storage. This is based on the assumption that a piece of free storage is obtained on the ZORCH subpool and the application is storing beyond the size of the obtained storage. By storing beyond the obtained storage, the application is overwriting a storage pointer used to keep track of unallocated storage within a partially allocated page. Storage management has a pointer within the page for every piece of unallocated storage on the page.

You can use the following STDEBUG and the CP TRACE commands to help you determine where in your application the user key pointers are being overlaid:

```
stdebug ZORCH (ob re stop
Ready;
ovtest1
14:45:54  * MSG FROM FRODO   : OBTAINED 00000008 013E5000 ZORCH    013E8036
TRACE STORE INTO 13E5008.8 CMD D 13E5008.8
B
 -> 00E125EA  ST    5040A004 >> 013E500C    CC 2
R013E5008  00000000 00000FE8                   E6
B
14:46:33  * MSG FROM FRODO   : OBTAINED 00000010 013E5FF0 ZORCH    013E806E
TRACE STORE INTO 13E6000.8 CMD D 13E6000.8
B
 -> 00E125EA  ST    5040A004 >> 013E500C    CC 2
R013E5008  00000000 00000FC8                   E6
B
14:47:21  * MSG FROM FRODO   : OBTAINED 00000020 013E5FD0 ZORCH    013E80A6
TRACE STORE INTO 13E5FF0.8 CMD D 13E5FF0.8
B
 -> 013E80A8  XC    D70F60006000 >> 013E5FF0    013E5FF0    CC 0
R013E5FF8  00000000 00000000                   E6
B
    013E80AE  XC    D70750005000 >> 013E5008    013E5008    CC 0
R013E5008  00000000 00000000                   E6
B
DMSFRO163E User key pointers have been destroyed (internal error code 85)
DMSFRM165I Chain header at address: 013FF5D0, Page address: 013E5000
DMSFRM817I Subpool name: ZORCH Subbk address: 013FFE34
```

To trap each call for storage on the ZORCH subpool, use the STOP option on the STDEBUG command. The STOP option places you into CP READ each time a call is made to obtain or release storage from the ZORCH subpool. When you enter CP READ, you set a CP TRACE STORE INTO for the 8-byte piece of storage that is immediately adjacent to the piece you just obtained. This 8-byte piece of storage is the storage management pointer.

The storage management pointer is the first 8 bytes of any unallocated piece of storage within the page. This storage management pointer contains the address of the next storage management pointer and the size of the unallocated piece of storage in which it is residing.

When one of the storage management pointers becomes corrupted, the `User key pointers have been destroyed` message is issued.

Analysis of the results shows us the following:

- A piece of free storage is obtained at X'13E5000' and isX'8' bytes in size. The trace is set at X'13E5008' for 8 bytes to trap a store in the storage pointer located there.

  The ST assembler instruction, trapped by the CP TRACE command, stores information into X'13E500C'. The address of the instruction storing into this field isX'E125EA'. This is storage management updating the size field in the storage pointer. This is a correct operation.
- A piece of free storage is obtained at X'13E5FF0' and isX'10' bytes in size. The trace is set at X'13E6000' for 8 bytes to trap a store into the piece of free storage beyond the piece obtained. Note that this address is in the page adjacent to the page where the storage is obtained. This page may even be on a different subpool. Stores into this location may not be in error as this may be valid piece of allocated storage.

  The ST assembler instruction, trapped by the CP TRACE command, stores information into the storage pointer at X'13E500C'. Again, by examining the address of the instruction, the store into this storage pointer is made by storage management updating the size field.
- A piece of free storage is obtained at X'13E5FD0' and isX'20' bytes in size. The trace is set at X'13E5FF0' for 8 bytes to trap a store into the piece of free storage beyond the piece obtained. Note that this is the address of the piece of free storage obtained previously. As long as that storage is still allocated, a store into it is a normal condition. If it does become unallocated, you are notified because you specified the RELEASE and the OBTAIN option on the STDEBUG command.

The XC assembler instruction, trapped by the CP TRACE command, stores information into allocated storage X'13E5FF0'. This was the application updating storage that was obtained. This is a correct operation.

- Finally, the following XC instruction attempts to store information at X'13E5008', which is a storage pointer:

```
    013E80AE   XC    D70750005000 >> 013E5008    013E5008    CC 0
```

  This XC instruction, which is not part of the CMS nucleus, has zeroed the storage pointer. You know this is unallocated storage because there never was an STDEBUG trace message issued for this address. It is most likely that the application using the piece of storage obtained at X'13E5000' for 8 bytes contains a bug. It is attempting to store beyond the piece of free storage it obtained.

By combining the STDEBUG and CP TRACE commands, you were able to locate the exact instruction that destroyed the free storage pointers.

Redirecting Console Messages Using the STDEBUG Command

You can use the USERID option on the STDEBUG command to direct the console messages to a user ID different from the user ID executing the application being traced. This can be useful when you have a second user ID and console. You can direct the messages to the second user ID and view them on console of that user ID. This leaves the console of the primary user ID, running the application, free for interaction. A STDEBUG trace can generate hundreds of messages when numerous calls are being made to storage management while the trace is active. When these messages become outstanding, waiting to be displayed on the console, it is often difficult to use PA1 or #CP to interrupt the program.

You can also use the USERID option to debug an application in a disconnected server machine. You can enable the storage management trace in the server machine using STDEBUG. The server machine can then direct the messages to an interactive user ID that can view the trace messages issued by the server.

Obtaining and Releasing Storage above 2 GB

CMS does not directly exploit storage above 2 GB. However, you can IPL z/Architecture CMS (z/CMS) in a virtual machine with more than 2 GB of storage, and programs running on z/CMS can use CMSSTOR OBTAIN and CMSSTOR RELEASE to allocate and return storage above 2 GB. Before assembling a program that uses storage above 2 GB, you must issue the GLOBAL MACLIB command to specify the DMSZGPI macro library ahead of the DMSGPI library. For example:

```
global maclib dmszgpi dmsgpi
```

To allocate storage above 2 GB, you must use SUBPOOL='USERG'. The number of pages of storage to be allocated is specified using the BYTES= parameter of CMSSTOR OBTAIN. For example, the following statement will allocate 1 page of storage above 2 GB:

```
CMSSTOR OBTAIN,BYTES=1,SUBPOOL='USERG'
```

The 64-bit address of the allocated storage is returned in general-purpose register 1. All storage allocated above 2 GB is aligned on a page boundary.

To release storage above 2 GB when z/CMS is running, you must specify SUBPOOL='USERG'. The ADDR= parameter must contain the 64-bit address of the storage to be released and the BYTES= parameter must contain the number of pages to be released. For example, the following statement will release 1 page of storage at the 64-bit address specified in register 10:

```
CMSSTOR RELEASE,BYTES=1,ADDR=(R10),SUBPOOL='USERG'
```

Chapter 7. Using Saved Segments

This chapter describes:

- What physical and logical saved segments are.
- How to use the SEGMENT macro to load, purge, and find the starting and ending address of a saved segment.
- How to use the SEGMENT command to reserve and release storage and to assign logical saved segments.
- How to use the QUERY SEGMENT command to display information about saved segments and segment storage spaces.

Physical and Logical Saved Segments

A ***saved segment*** is an area of virtual storage that is assigned a name, loaded with data or programs, then saved in a system data file in spool space. A saved segment can be attached to and detached from a virtual machine. Using saved segments is a way of using storage that is not yours.

Segment spaces, member saved segments, and discontiguous saved segments (DCSSs) reside on CP-owned volumes and must be defined to CP before being used. A segment space, which begins and ends on a megabyte boundary, contains one or more member saved segments, which begin and end on page boundaries. A DCSS also begins and ends on a megabyte boundary, but does not contain members.

Defining frequently-used data or programs as saved segments provides several advantages:

- Many users can access the same saved segment, which helps you use real storage more efficiently.
- Saved segments need not be in the address range of a virtual machine (this can also help you use storage more efficiently).
- Space for saved segments can be reserved within a virtual machine's address space, which helps you make sure that the saved segment is always available.

A **physical saved segment** is a member saved segment or DCSS that may contain one or more **logical saved segments** that CMS recognizes. Defining logical saved segments provides further advantages:

- Each logical saved segment can contain different types of program objects, such as modules, text files, execs, callable services libraries, language information, and user-defined objects, or a single minidisk file directory. You can use logical saved segments to package your entire application. For example, you may want to create a logical segment definition file that defines the parts of your application. You could then send it to the system administrator, who will create the logical saved segment and make it available for others to use.
- You can use physical saved segments more efficiently by defining many different logical saved segments in a single physical saved segment.
- Users can access specific logical saved segments rather than all the contents of a physical saved segment.

For information about defining saved segments, see *z/VM: Saved Segments Planning and Administration*.

Using the SEGMENT Macro

You can use the SEGMENT macro to:

- Load a saved segment into storage
- Find the start and end address of a saved segment
- Purge a saved segment.

Loading a Saved Segment

Use the SEGMENT LOAD macro to load a saved segment. SEGMENT LOAD reserves a storage space, if one is not already reserved, and loads the saved segment into it.

Example 1

To load a shared copy of a saved segment named MYSEG that survives abend processing, code:

```
SEGMENT LOAD,NAME=MYSEG,SYSTEM=YES,SHARE=YES
```

Note that SHARE=YES is a default value and can be omitted.

Example 2

To load a private copy of a saved segment named MYSEG that does not survive abend processing, code:

```
SEGMENT LOAD,NAME=MYSEG,SYSTEM=NO,SHARE=NO
```

Note that SYSTEM=NO is a default value and can be omitted.

Table 17 on page 76 describes some of the return codes stored in register 15 when the SEGMENT LOAD macro completes. For a complete list, see the *z/VM: CMS Macros and Functions Reference*.

Table 17. Return Codes for SEGMENT LOAD Macro

Contents	Meaning
0	The saved segment was loaded successfully; register 1 contains the beginning address of the loaded saved segment and register 0 contains the highest address of the loaded saved segment.
12	The saved segment was already loaded.
41	The storage required to load the saved segment is already in use.
44	The saved segment does not exist.
256	An error occurred while processing the contents of a logical saved segment.

For any nonzero return code other than 12, registers 0 and 1 are set to zero.

Determining the SHARE Attribute

The SHARE attribute of a physical saved segment that contains logical saved segments is determined by the first logical saved segment that you load. When you load subsequent logical saved segments defined in the same physical saved segment, you must specify (or default to) the same attribute.

For example, suppose physical saved segment USERSEG contains logical saved segment MYSEG. If you specify SHARE=NO when you load MYSEG, you must specify SHARE=NO when you load any other logical saved segments in USERSEG. If the SHARE parameter does not match the SHARE attribute of the physical saved segment, the saved segment is ***not*** loaded and register 15 contains a return code of 36.

See "How CMS Handles Objects in Logical Saved Segments" on page 78 for information regarding the way which CMS handles objects in logical saved segments.

Finding the Starting and Ending Address of a Saved Segment

Use the SEGMENT FIND macro to find the starting and ending address of a saved segment. For example, to find the starting and ending address of MYSUB, code:

```
SEGMENT FIND,NAME='MYSUB'
```

If the saved segment exists, its starting address is stored in register 1; its ending address is stored in register 0.

Table 18 on page 77 describes some of the return codes stored in register 15 when the SEGMENT FIND macro completes. For a complete list, see the *z/VM: CMS Macros and Functions Reference*.

Table 18. Return Codes for SEGMENT FIND Macro

Contents	Meaning
0	The saved segment has not been loaded.
12	Indicates the saved segment has already been loaded.
44	The saved segment does not exist.

Purging a Saved Segment

Use the SEGMENT PURGE macro to detach a saved segment from your virtual machine. For example, to remove MYSUB, code:

```
SEGMENT PURGE,NAME='MYSUB'
```

Purging a logical saved segment removes the objects in the saved segment from use by CMS. If MYSUB is a logical saved segment and is the only loaded or reserved logical saved segment within the physical saved segment, then the physical saved segment is detached from the virtual machine. If the physical saved segment is a member of a CP segment space and is the only loaded or reserved member in that segment space, then the segment space is detached from the virtual machine. The reserved segment storage space is also released unless it was explicitly reserved using the SEGMENT RESERVE command.

When you use SEGMENT PURGE to purge a saved segment, the saved segment must have been loaded using SEGMENT LOAD. If the saved segment was loaded using the DIAGNOSE code X'64' LOADSYS function, you must use the DIAGNOSE code X'64' PURGESYS function to purge the saved segment. You cannot use SEGMENT PURGE and the DIAGNOSE code X'64' PURGESYS function interchangeably.

Table 19 on page 77 describes some of the return codes stored in register 15 when the SEGMENT PURGE macro completes. For a complete list, see the *z/VM: CMS Macros and Functions Reference*.

Table 19. Return Codes for SEGMENT PURGE Macro

Contents	Meaning
0	The saved segment was successfully purged.
40	The saved segment was not loaded.
44	The saved segment does not exist.
256	An error occurred while processing the contents of a logical saved segment.

How CMS Locates Saved Segments

CMS uses the following process to locate a saved segment to be loaded:

1. CMS searches the list of logical saved segments for one with the name specified on the SEGMENT LOAD macro. If a logical saved segment is found, a storage space for the associated physical saved segment is reserved (if not already reserved). If the physical saved segment is a DCSS, the storage space is reserved for the DCSS. If the physical saved segment is a member of a CP segment space, the storage space is reserved for the entire segment space. Then the storage space is loaded (if not already loaded), and the contents of the logical saved segment are processed.
2. If a logical saved segment with the specified name is not found, CMS searches the list of storage spaces previously reserved with the SEGMENT RESERVE command to determine if a space has been reserved for a saved segment with the requested name. If one is found, the storage space is loaded (if not already loaded).

3. If no reserved storage space exists, CMS determines whether the requested saved segment has been defined in CP. If so, CMS issues a SEGMENT RESERVE command to create a reserved storage space, then loads the saved segment. If the saved segment is a member of a CP segment space, CMS reserves storage for and loads the entire segment space.
4. If the requested saved segment is none of the above, the appropriate return code (RC=44) is returned to the calling program.

CMS uses the same search order to find or purge a saved segment.

How CMS Handles Objects in Logical Saved Segments

When a logical saved segment is loaded, the MODULE or TEXT files contained within it are established as nucleus extensions or subcommand processors, execs are established as EXECs-in-storage, callable services libraries are available for the GLOBAL CSLLIB and RTNLOAD commands, application language information is activated, and user object load routines are called.

All application language information whose languages match the current system language is added to the active set of applications. Thus, the application does not need to issue a SET LANGUAGE command with the ADD option. When a SET LANGUAGE command is issued that changes the current system language, all application language information for the old language is dropped, and any language information that matches the new system language and is in a currently loaded logical saved segment is automatically added.

Logical saved segments and the objects in them are loaded last-in-first-out (LIFO). Nucleus extensions, subcommand processors, and execs (all of which are in a saved segment that has been loaded) override previous definitions with the same name. To reactivate previous definitions, you can drop saved segment-resident nucleus extensions (using NUCXDROP) or execs (using EXECDROP), or purge the saved segment. After an object in a saved segment has been dropped, the saved segment must be purged and reloaded to reactivate the object.

Objects in other logical saved segments within the physical saved segment are not processed.

Using the SEGMENT Command

You can use the SEGMENT command to perform the following additional functions not provided in the SEGMENT macro:

- Reserve storage space for a saved segment
- Release the storage that was reserved
- Assign a logical saved segment to a physical saved segment.

Reserving Storage Space for Saved Segments

In CMS, saved segments can be located within your virtual machine's address space. For saved segments that are **not** loaded immediately after IPL, you should consider reserving storage space for the saved segment. If you do not reserve space for the saved segment, other programs can use the storage. If the required storage is occupied when you try to load a saved segment, the load fails.

You can use the SEGMENT RESERVE command to reserve space for saved segments. Reserving space for saved segments (a) allows you to ensure that your applications can load the saved segments in the storage they specify and (b) eliminates the possibility of saved segments overlaying or being overlaid by portions of CMS. For example, to reserve space for a saved segment named MYSEG, enter:

```
segment reserve myseg (system
```

The SYSTEM option specifies that the space reserved will not be released if abend processing occurs.

Releasing Segment Storage Spaces

The SEGMENT RELEASE command releases the storage that has been reserved for a saved segment or reclaims storage where saved segments have been loaded.

For example, to release storage for MYSEG, enter:

```
segment release myseg
```

SEGMENT RELEASE uses the following process to release storage:

1. If the specified saved segment is a logical saved segment, it is removed from the list of reserved logical saved segments. If the physical saved segment that contains the logical saved segment no longer has any logical saved segments loaded or reserved, the physical saved segment is detached from your virtual machine and the reserved storage is returned to CMS (that is, the physical saved segment is released). If the physical saved segment is a member of a CP segment space, and the segment space no longer has any members loaded or reserved, the segment space is released and the storage is returned to CMS.
2. If the specified saved segment is a physical saved segment, all the loaded or reserved logical saved segments within the physical saved segment are released first, then the physical saved segment is released, then (if applicable) the CP segment space is released and the storage is returned to CMS.
3. If the specified saved segment is a CP segment space, and if any members of the segment space are physical saved segments that contain logical saved segments, all the loaded or reserved logical saved segments are released first, then the members of the segment space are released, then the segment space is released and the storage is returned to CMS.

Assigning Logical Saved Segments to Physical Saved Segments

Use the SEGMENT ASSIGN command to assign or associate a logical saved segment with a physical saved segment. When the name of a logical saved segment is associated with two physical saved segments, the default logical saved segment is the last one in the system segment identification file (SYSTEM SEGID S2). You can change the default association by using the SEGMENT ASSIGN command. **Do not use any other method to modify this file.**

To associate a logical saved segment named APPLSEG to the physical saved segment named MYSEG, enter:

```
segment assign applseg myseg
```

For more information on the SEGMENT command, see the *z/VM: CMS Commands and Utilities Reference*.

Displaying Information about Saved Segments

Use the QUERY SEGMENT command to display information about loaded or reserved saved segments and the storage spaces that contain or are reserved for saved segments. For example,

```
query segment nlsuceng
```

might return a response similar to the following:

```
Space    Name     Location  Length    Loaded  Attribute
NLSUCENG NLSUCENG 00DA0000  00100000  YES     USER
```

To display information on all the currently loaded or reserved saved segments, enter:

```
query segment *
```

In response, CMS returns something similar to the following:

```
Space     Name      Location  Length    Loaded   Attribute
NLSUCENG  NLSUCENG  00DA0000  00100000  YES      USER
```

```
PSEG2     EXECSEG   02000000  0003F000  NO        SYSTEM
PSEG2     MYSEG     02240000  00380000  NO        SYSTEM
```

For a logical saved segment, the Space column displays the name of the physical saved segment in which it resides, and the Name column displays the name of the logical saved segment. For an explicitly-loaded or reserved physical saved segment, the Space column displays the name of the storage space in which it resides, and the Name column displays the name of the physical saved segment. If the physical saved segment is a DCSS, the storage space is the DCSS. If the physical saved segment is a member of a CP segment space, the storage space is the CP segment space. The Loaded column indicates whether the saved segment has been loaded or just reserved.

To display the contents of a logical saved segment, use the CONTENTS option. For example, to display the contents of the APPLSEG logical saved segment, enter:

```
query segment applseg contents
```

The response is in the following form:

```
Type      Location  Length    Name
NUCEXT    006E0630  00000038  TESTMOD1
SUBCOM    006E0F18  00000038  TESTMOD2
EXEC      006E32D0  00000848  PROF1    EXEC
EXEC      006E0698  00000848  TEST     XEDIT
LANGUAGE  006E3030            AMENG    OFS
CSL       006E0000  00000610  LIB2
USER      006E3B50  000000FF  TESTUSER
```

To display the contents of the USERDISK logical saved segment that contains a saved minidisk directory, enter:

```
query segment userdisk contents
```

The response is in the following form:

```
Type      Location  Length    Name
DISK      00DA5000  00010000  LABEL1
```

Use the ASSIGN option to display the physical saved segment to which a logical saved segment is currently assigned, as follows:

```
query segment lseg1 assign
```

The response is in the following form:

```
Lsegname  Assignment
LSEG1     PSEG1
```

Use the SPACE option to display information about segment storage spaces. To display information about all segment storage spaces, enter:

```
query segment * space
```

The response is in the following form:

```
Space     Name      Location  Length    Loaded    Attribute
NLSUCENG  -         00DA0000  00060000  YES       -
PSEG2     -         02000000  01000000  NO        -
```

The Space column contains the name of the segment storage space, the Name column always contains a dash (-), the Location column contains the starting address, and the Length column contains the length of the storage space.

Note the difference between QUERY SEGMENT * and QUERY SEGMENT * SPACE. QUERY SEGMENT * lists all the currently loaded or reserved segments, which can include logical saved segments, explicitly-loaded physical saved segments, or CP segment spaces. On the other hand, QUERY SEGMENT * SPACE lists all the segment storage spaces that contain or are reserved for saved segments. In the previous response to QUERY SEGMENT *, NLSUCENG is a segment space that has been explicitly loaded, and PSEG2 is a physical saved segment that contains the logical saved segments EXECSEG and MYSEG, each of which has been reserved but not loaded.

For more information on the QUERY SEGMENT command, see the *z/VM: CMS Commands and Utilities Reference*.

Chapter 8. Console and Terminal I/O

This chapter describes how to do terminal and console I/O. In particular, it describes how to use the following macros:

- CONSOLE — Performs full-screen I/O operations.
- APPLMSG — Retrieves and displays messages.
- LINERD — Reads one or more lines of input.
- LINEWRT — Writes one or more lines of output.

Performing 3270 Full-Screen I/O Operations

The CMS CONSOLE macro provides an assembler language interface to 3270 full-screen I/O operations. Using the CONSOLE macro allows your applications to become less dependent on low-level system architecture (in particular, architecture-dependent I/O instructions).

Before the CONSOLE macro existed, an application had to (a) construct data streams, (b) build the channel command word (CCW), (c) determine the type of device (dedicated devices or virtual console) and set up for DIAGNOSE code X'58' or a 3270 SSCH instruction, and (d) check the channel status word (CSW or SCSW) to determine what action should be taken.

Using the CONSOLE macro, your application needs only to build a data stream, issue the appropriate CONSOLE macro call, and check a return code. The 3270 data stream is made up of a command code, followed by a combination of WCC, orders, and data or structured fields, depending on the exact nature of the screen transaction. The command code is defined by the OPTIONS parameter, the rest of the data stream should be placed in the area defined by the BUFFER parameter. (See the *IBM 3270 Information Display System Data Stream Programmer's Reference* for details of how to construct a data stream.)

CONSOLE Macro Functions

The basic functions of the CONSOLE macro are:

- CONSOLE OPEN — Opens a specific path to a device.
- CONSOLE CLOSE — Closes a specific path to a device.
- CONSOLE MODIFY — Change, delete or add certain parameters of a previous path definition.
- CONSOLE READ — Reads information from the display device.
- CONSOLE WRITE — Writes buffers that have 3270 data streams built by the application.
- CONSOLE WAIT — Waits for an interrupt (for example, an I/O interrupt from the console device).
- CONSOLE QUERY — Gets information about the device attributes or about a specific path and its associated device (if the path is open). This information is the same as that returned by DIAGNOSE codeX'24' and DIAGNOSE code X'8C' plus some additional information.
- CONSOLE EXCP — Lets you specify your own channel program to read or write I/O. Note that CONSOLE EXCP requires you to distinguish between System/370 mode and 370-XA or ESA/XC modes of operation. It also requires you to distinguish between dedicated devices and the virtual console, because DIAGNOSE code X'58' CCWs must be provided for I/O to the virtual console.

  Note: Only CMS levels prior to CMS Level 12 will execute in a 370 virtual machine.

Opening a Path to a Console

To use the CMS console facility for I/O, you must first open a path to a device. Opening a path associates a unique path name with your application and the device you want to use.

Use CONSOLE OPEN to define a path to a device. For example, the following CONSOLE macro:

```
CONSOLE OPEN,PATH='PATH1',EXIT=HANDLER,UWORD=INFO,BUFFER=(BUFF), *
      ERROR=ERROR1
```

- Opens a path named PATH1 (PATH=‘PATH1’) to the virtual console. (To specify a device other than the virtual console, for example, a dialed device or a 3270 graphics printer, specify DEVICE=*vdev* where *vdev* is the address of the device.)
- Defines HANDLER as the entry point of the routine that handles unsolicited interrupts from the console (EXIT=HANDLER). See “Handling Console Interrupts” on page 84 for more information about using the EXIT parameter to define an interrupt handler.
- Passes in register 0 the contents of the fullword located at INFO to the interrupt handler routine (UWORD=INFO). You can use UWORD to pass a control block address, a routine address, or anything else you want to pass to the exit routine.
- Defines (BUFF) as the address of a storage location where CMS stores information about the path and its associated device (BUFFER=(BUFF)). You can use the CQYSECT macro to map this area. Also note that you can use the CONSOLE QUERY macro to obtain information about the path and device. You can use this information later when you build your data stream.
- Defines ERROR1 as the entry point of a routine that gets control if CONSOLE returns a nonzero return code (ERROR=ERROR1). Your error routine can examine the return code and determine if more processing can be performed (for example, a return code of 28 specifies that the path is already open).

A parameter not shown in the example above is RESET. Specifying RESET=YES will cause a CP RESET command to be issued when the path is the last path to a dedicated device and the path is being deleted. RESET=YES is the default. The RESET parameter is ignored if the device is the virtual console. If RESET=NO is specified, an application must issue a CP RESET to free up the device.

For a CONSOLE OPEN request when the path already exists, device information will be obtained for the associated device. The information is updated in the existing device entry if different from the original device information. A return code of 28 will be set and the device information will be returned in a buffer, if the application provided one.

For a typewriter-type device (TTY), a console path is not opened and device characteristics are not saved in a console device entry. However, if the application provides a buffer, device information is returned in the buffer.

Handling Console Interrupts

Normally, you do not have to do anything special to handle console interrupts. If the interrupt is solicited (for example, your program issues either a CONSOLE WAIT or a CONSOLE READ with the WAIT parameter), CMS passes control to the code generated by the CONSOLE macro, which then processes the interrupt.

Two examples of when you may want to handle unsolicited interrupts include:

- When you need to handle unsolicited attention interrupts; for example, when a user inadvertently enters more information than the application expects. The application may want to stack the unsolicited attentions, and then do a CONSOLE READ to clear the interrupts later.
- When an application program works with dedicated 3270 devices. For example, some 3270 devices receive an unsolicited device end (DE) when the device is powered on. Your exit routine can do a CONSOLE QUERY to determine that the device has been powered on; it can then post an event control block (ECB) or issue an EventSignal to notify the main program that the device is ready. Note that exits cannot enable for interrupts and, therefore, should not do their own I/O to the device.

To handle unsolicited interrupts for a 3270 device, you can use the EXIT parameter of CONSOLE OPEN to define an exit routine. If you do not define an exit routine, the interrupt is handled by CMS interrupt handling routines.

Note: To avoid confusion, do not use the CONSOLE macro and the HNDIO, HNDINT, or OS/MVS STAX macros to define exit routines for the same device. Code generated by the CONSOLE macro handles solicited console interrupts; however, for unsolicited interrupts, exit routines defined by HNDIO, HNDINT, or OS/MVS STAX supersede those defined by the CONSOLE macro. (For a discussion of how CMS handles unsolicited interrupts and a more thorough discussion of CMS interrupt handling in general, see "Handling I/O Interrupts" on page 179.)

Console Exit Routine Entry Conditions

If you do specify an exit routine, your program must be prepared to handle the interrupt. The exit routine receives control as an extension of CMS I/O interrupt handling; the PSW is set up with a system storage key and is disabled for interrupts.

After full-screen mode has been established in your virtual machine using the CONSOLE macro, only attention interrupts will be passed to exits for the virtual console. Exits defined for dedicated 3270 devices, however, will receive other interrupts as well (such as device ends).

When the screen is put into CP console mode (line mode) with a 3215 SIO issued by CMS to the virtual console, attention interrupts will be given to the CMS second level interrupt handler (SLIH) instead of a full-screen CONSOLE exit until full-screen mode is reestablished using the CONSOLE macro. After the virtual console is back into full-screen mode, the CONSOLE exit can then be given attention interrupts again.

When your exit routine receives control, the significant registers are set up as follows:

- Register 0 contains a fullword of information that the user can specify on the UWORD parameter.
- Register 13 contains the address of a register save area that the exit routine can use.
- Register 14 contains the return address.
- Register 15 contains the exit routine's entry point address.

The addressing mode (AMODE) of the exit routine is the same as the addressing mode of the program that issued the CONSOLE OPEN or CONSOLE MODIFY to define the exit routine.

When an attention interrupt is passed to your exit routine and the interrupt is not eventually handled by a read operation, the virtual console may be delayed.

The BRKKEY parameter of the CONSOLE WRITE function allows a fullscreen application to specify whether or not the break key interrupt is reflected to the application. If break key handling is requested and CP posts an attention to the virtual machine, the break key is passed to the application when a fullscreen read is issued. This replaces the normal break key function of returning the virtual machine to CP mode. (For more information see the *z/VM: CMS Macros and Functions Reference*)

To obtain status information about the interrupt or the interrupting device, you can issue CONSOLE QUERY specifying the PATH parameter and providing a buffer where CMS can store the information. CONSOLE QUERY returns the CSW/SCSW, the sense data (if any), the last CCW executed, and various device information.

Modifying Parameters of a CONSOLE OPEN

Use CONSOLE MODIFY to change parts of the definition for a path to a device. You can change or delete the settings of the EXIT, UWORD or RESET parameters of a previously issued CONSOLE OPEN. This lets an application dynamically modify these parameters, without having to close and reopen the console path with new values.

If you have not specified EXIT, UWORD or RESET in a CONSOLE OPEN, you can initially set these parameters with a CONSOLE MODIFY. As with OPEN, the exit routine will be established in the same addressing mode (AMODE) as the application issuing the CONSOLE MODIFY call. The UWORD parameter only has meaning if an exit routine has been established. An application must have already opened a path before issuing a CONSOLE MODIFY request on it.

Writing to and Reading from a Console

Use CONSOLE WRITE to write a 3270 data stream and CONSOLE READ to read information from the screen. For all applications that use the CONSOLE macro, CMS keeps track of which application ‘owns’ (performed the last I/O operation to) each display screen. If one CONSOLE macro application currently owns the screen and your CONSOLE macro application wants to perform I/O, your application needs to gain control of the screen by reformatting it with an erase/write (EW), an erase/write alternate (EWA), or a write structured field (WSF) when a WSF buffer contains structured fields for an EW or EWA operation.

Note: CMS cannot coordinate I/O for applications that modify PSWs in low storage and issue their own DIAGNOSE code X'58' instructions. Until all applications use the CONSOLE macro instead of DIAGNOSE code X'58', you may see data from two different applications mixed on the screen. For more information, see “Notes on the CONSOLE Macro” on page 89.

CONSOLE WRITE Options

You can use the parameters of CONSOLE WRITE to specify what type of a write operation you want to perform:

- CLEAR — Clears the physical screen before the buffer (if any) is written. To clear the screen, specify CLEAR without the BUFFER parameter.
- NOCLEAR — Does not clear the screen. The user at the terminal may need to clear the screen before CMS writes the buffer (similar to the MORE...condition). NOCLEAR is the default value.
- W — Writes the buffer with an ordinary Write command, overlaying the current contents of the display screen. If you do not specify W, EW, EWA, or WSF, then W is assumed.
- EW — Writes the buffer with the Erase/Write option.
- EWA — Writes the buffer with the Erase/Write Alternate option to establish the alternate screen mode for the device.
- WSF — Writes the buffer with the Write Structured Field option to provide control information to the device.
- (*reg*) – Specifies a register 2-12, whose low-order byte contains the option or options to be used.

Note: You must specify the BUFFER parameter for all options other than CLEAR. For the complete syntax of the CONSOLE macro, see the *z/VM: CMS Macros and Functions Reference*.

CONSOLE READ Options

Use CONSOLE READ to read from a display device. The parameters on CONSOLE READ let you specify:

- WAIT — Specifies that if an interrupt has not been received from the device since the last write operation completed, processing of this request is suspended until such an interrupt occurs. When the interrupt is received, the READ is performed. This is the default.
- NOWAIT — Specifies that this read request is to be processed immediately. This lets you read the inbound data stream from the device without waiting for input.
- RDMOD — Processes this request as Read Modified to transmit only the modified fields from the screen. This is the default.
- RDBUF — Processes this request as Read Buffer to transmit the entire contents of the screen.
- (reg) — Specifies a register 2-12, whose low-order byte contains the option or options to be used.

Waiting for Console Interrupts

To cause your program to wait for console interrupts, you can (1) use the WAIT parameter of the CONSOLE READ macro, or (2) issue a CONSOLE WAIT macro. For performance reasons, it may be better to use the WAIT parameter of CONSOLE READ, because it involves fewer SVC calls than if you issue a CONSOLE READ and a CONSOLE WAIT.

The Console facility loads an enabled wait PSW to receive input that can be read by the application. The remainder of the Console facility runs disabled for interrupts. Applications that do not want to handle unsolicited interrupts should be disabled in their routines that perform I/O using the Console facility; or at least during their CONSOLE macro calls.

If you use CONSOLE WRITE to write data to a display that is disconnected, you receive a return code of 1. If you receive a return code of 1 and you **do** want to wait for a response, you must issue CONSOLE WAIT rather than using the WAIT parameter of CONSOLE READ. CONSOLE READ does not wait for an interrupt from a disconnected terminal; instead, it returns the error code of 1 to the application.

Example

To use CONSOLE WAIT to wait for an interrupt from the device on PATH1, you could code:

```
CONSOLE WAIT,PATH='PATH1'
```

You can use the ERROR parameter of CONSOLE WAIT if you want to define a separate routine to examine nonzero return codes.

Completing an I/O Operation

When an I/O operation completes, your application can check the return code in register 15 to make sure it should continue processing. Also, if you are using CONSOLE READ, register 0 will contain the length of the data read.

Before it passes a return code back to your application, the CONSOLE macro (a) does extensive CSW/SCSW checking and (b) retries I/O operations when necessary. This status checking should be sufficient for most programs. If, however, you want more information about the I/O just performed, issue CONSOLE QUERY specifying the PATH parameter. The CONSOLE QUERY shows the CSW/SCSW after I/O, the sense data (if any), the last CCW executed, and all the device information from the device entry.

Obtaining Information about a Console Path

As mentioned earlier, if you specify the BUFFER parameter of CONSOLE OPEN, CMS returns to the buffer information about the path and the device associated with it.

You can also use CONSOLE QUERY to obtain information about a path and its associated device. CONSOLE QUERY can also return information about virtual devices that do not yet have paths associated with them.

Example 1

To determine if a path named PATH1 exists, code:

```
CONSOLE QUERY,PATH='PATH1'
```

If the path does not exist, CMS returns a return code of 28 in register 15.

Example 2

To obtain information about the device associated with PATH1, you could code:

```
CONSOLE QUERY,PATH='PATH1',BUFFER=(BUFF)
```

Example 3

To obtain information about device 000A, regardless of whether 000A is associated with a path, you could code:

```
CONSOLE QUERY,DEVICE=000A,BUFFER=(BUFF)
```

Note that if device 000A is not defined, you get a return code of 40 in register 15.

Using a Buffer

In the previous two examples, BUFFER=(BUFF) specifies the address of a storage area where CMS stores information. You can use the CQYSECT macro to map this area. If the buffer length is less than the length of CQYSECT, the data in the buffer is truncated. CONSOLE QUERY stores into register 0 the length of data it actually moves into the buffer. CQYSECT provides length values you can use to specify the size of the buffer. For more information on the CQYSECT macro, see *z/VM: CMS Macros and Functions Reference*.

A CONSOLE OPEN and CONSOLE QUERY DEVICE will obtain new device information and update a device entry for that device if one already exists. A CONSOLE QUERY, however, will extract whatever information is in the path and device entries. This will reflect the state of the path and associated device either when the path was opened or at the time of the last I/O if I/O was performed for that path. This also avoids the overhead of DIAGNOSE instructions when CONSOLE QUERY PATH is issued from exit routines driven by the I/O first level interrupt handler. Therefore, when disconnecting and reconnecting to another device, there is a possibility the original device information will be returned when a CONSOLE QUERY PATH is done.

Disconnecting and Reconnecting

If the Console Facility has not yet determined if a disconnect and reconnect to another device occurred, a CONSOLE QUERY PATH may reflect the original device information. A CONSOLE OPEN and QUERY DEVICE will update a device entry, and I/O error processing may update a device entry. Also, when an I/O interrupt occurs, a field is checked that CP may have updated if a disconnect and reconnect has occurred. In this case, as well as when an I/O error occurs, new device information is obtained and the information is compared to the existing device entry information. If different, the device entry is updated and a return code 2 is passed back to the application indicating that device characteristics have changed and another query may be necessary. Depending on the device, CP may or may not give back an I/O error to the console facility, depending on whether the device can handle the given data stream.

Writing Your Own Channel Programs

If a program needs to build its own CCWs, you can use CONSOLE EXCP to do so. CONSOLE EXCP does not attempt to validate or convert the channel program. Because CONSOLE EXCP does not check the CCW, the console facility cannot determine the type of I/O requested and, therefore, cannot coordinate the screens as effectively as with CONSOLE READ or CONSOLE WRITE functions.

CONSOLE EXCP is not recommended for the virtual console because (a) you must know the internal implementation of how the I/O is issued and (b) it requires a knowledge of the virtual machine architecture. You can, however, use CONSOLE EXCP to chain several CCWs. The first CCW in the string should be an EW, EWA, or WSF to reformat and to gain control of the screen.

Example 1

To use a CCW that your System/370 application has already built, you could code CONSOLE EXCP as follows:

```
CONSOLE EXCP,PATH='PATH1',CCW=CCWADDR
```

PATH=‘PATH1’ specifies the name of the path. CCW=CCWADDR specifies the address of a channel program, which consists of one or more format-0 CCWs that indicate the operation(s) to be performed.

Example 2

To use a CCW that your XA or XC application has already built, you could code CONSOLE EXCP as follows:

```
CONSOLE EXCP,PATH='PATH1',ORB=ORBADDR
```

PATH=‘PATH1’ specifies the name of the path. ORB=ORBADDR specifies the address of an Operation Request Block (ORB) for a channel program. The ORB can indicate either format-0 or format-1 CCWs depending on whether you are addressing data above or below the 16MB line. For CCWs below 16MB, you can use format-0 or format-1. For CCWs above 16MB, you must use format-1. CCWs for data above 16MB. The ORB must contain a pointer to a channel program for the operation(s) to be performed.

Note:

1. The ORB parameter is invalid for paths to the virtual console and in a 370 virtual machine.
2. Command codes for the DIAGNOSE code X'58' must be provided if you are doing I/O to the virtual console. Otherwise, command codes defined for the dedicated device should be used in the CCW. For dedicated devices, a SSCH is issued.
3. Do not specify the ORB and CCW parameters on the same macro call.
4. If you specify the CCW parameter, the Console facility assumes format-0 when setting up the ORB.

Closing a Console Path

Use CONSOLE CLOSE to close a path to a device. CMS automatically closes paths during abend processing, and when the device is detached by a CP DETACH command or redefined by a CP DEFINE or CP REDEFINE command.

Example

To close PATH1, you could code:

```
CONSOLE CLOSE,PATH='PATH1'
```

Note:

1. If RESET=YES is specified when a path is opened or modified, then a CP RESET command is issued when you close the last path to the dedicated 3270.
2. A return code of 3 means that the requested path is closed, but other paths remain open to the device. You can use the ERROR parameter of CONSOLE CLOSE if you want to define a separate routine to examine nonzero return codes.

Notes on the CONSOLE Macro

1. CMS cannot coordinate I/O for applications that modify PSWs in low storage and issue their own DIAGNOSE code X'58' instructions. Until all applications use the console facility instead of DIAGNOSE code X'58', you may see data from two different applications mixed on the screen. If mixed data does appear on the screen, you can do one of the following (as appropriate):
   a. Your application can specify the EW or EWA options of CONSOLE WRITE to gain control of the screen.
   b. You can press the CLEAR key or issue a command that causes an erase/write.
   c. For applications running in full-screen CMS, you can write an EXEC to (a) issue a SET FULLSCREEN SUSPEND command, (b) invoke the full-screen program, and, when processing completes, (c) resume full-screen CMS by issuing a SET FULLSCREEN RESUME.
   d. When you issue CONSOLE WRITE, if CP breaks in and writes a screen or if another application had previously used the CONSOLE macro to write a screen, you will receive a return code of 32. If you get return code 32, respecify CONSOLE WRITE using the EW (erase/write) or EWA (erase/write alternate) options.

      Note that if you have issued a CONSOLE READ and CP breaks in and writes a screen (such as a CP warning message), the read is performed and a channel end/device end is returned to CMS. The Console facility gives your application a return code of 0. Your application should examine the data stream attention identification (AID) to determine whether there are any modified fields to process. An AID byte of X'60' indicates no operation or an unsolicited attention. CP will likely return X'8E' in the unit status byte of the CSW/SCSW (causing the CONSOLE macro return code 32) on the next full-screen write.

2. If you are using the CONSOLE EXCP function or a CONSOLE WRITE with the WSF option, the CONSOLE facility does not know what operation is being requested because it does not scan the buffer or verify the CCWs. Because of this, the application may need to do more screen coordination of its own. For example, a write will be issued regardless of the fact that another path last wrote to the screen. Either an EW/EWA can be done to ensure a complete erasure of the screen, or, a CONSOLE QUERY PATH can be issued to check the CQYPLIO field. This will tell if your path was the last path to do I/O using the CONSOLE macro, and whether you need to do an EW/EWA.
3. Applications that need to coordinate line mode and full-screen I/O should use the WAITT macro before they issue the CONSOLE macro. WAITT waits for any pending line mode I/O to complete.
4. Many full-screen applications run disabled for interrupts in their I/O routines so they will not receive unsolicited interrupts at that time. This should also be done in applications using the Console facility so an interrupt does not come in when CONSOLE is returning to the application, particularly if the application has not established their own interrupt handling routine. This is important for WSF operations that generate attention interrupts back to CMS.

A Sample Program

The following program shows how to code a simple full-screen program using the CONSOLE macro. The user must clear the screen after...MORE appears in the status area and press ENTER after each screen of output is displayed. A return code of 0 or 3 indicates a normal return.

A Sample Program Using the CONSOLE Macro

```
SCT      CSECT
         LR    12,15
         USING SCT,12
*
*        Open path "SCT" to the virtual console
         CONSOLE OPEN,PATH='SCT',ERROR=OPENERR
*
*        Set up registers to point to the first buffer
         LA    R5,BUFFERS            * address of first buffer
         L     R7,0(R5)              * point to the first
                                       buffer
         LA    R6,LENGTHS            * address of 1st buffer
                                       length
         L     R8,0(R6)              * point to the 1st buffer
                                       length
*
* Write first screen (need to do an ERASE WRITE to get control)
         CONSOLE WRITE,PATH='SCT',BUFFER=((R7),(R8)),BRKKEY=NO,
               OPTIONS=(EWA),ERROR=WRITEERR*
*
* R4 stores how many buffers are left to print.
         LA    R4,NUMBUFFS          * space to store addrs of
                                      buffs
         SRL   R4,2                 * divide by four
                                      (4 byte addr)
         BCTR  R4,R0                * subtract 1; first write
                                      done
*
*        Wait for an interrupt before showing the next screen
*        and clear the "ENTER" with the READ
WRITE    EQU   *
         CONSOLE READ,PATH='SCT',OPTIONS=(WAIT),
               BUFFER=(READBUF),ERROR=READERR *
*
* Write the remaining screens by incrementing the registers
* and using the WRITE option (not ERASE/WRITE)
         LA    R5,4(R5)            * point to the next
                                     buffer address
         L     R7,0(R5)            * point to the next buffer
         LA    R6,4(R6)            * point to next buffer
                                     length addr
         L     R8,0(R6)            * point to the next buffer
                                     length
         CONSOLE WRITE,PATH='SCT',BUFFER=((R7),(R8)),  *
               OPTIONS=(W),ERROR=WRITEERR
         BCT   R4,WRITE            * decrement count and
                                     branch to next write
*
```

```
* Wait for an interrupt after showing the last screen
         CONSOLE READ,PATH='SCT',OPTIONS=(WAIT),
         BUFFER=READBUF, ERROR=READERR*
*
* Close path "SCT"
DONE     EQU   *
         CONSOLE CLOSE,PATH='SCT'
         CMSRET
* Error Routines
OPENERR  EQU   *
         LR    R9,R15
         APPLMSG TEXT=&apos.OPEN ERROR
                     RC=&&1&apos.,APPLID=CMS,  *
         SUB=(DEC,((9),4))
         BR    R14
WRITEERR EQU   *
         LR    R9,R15
         APPLMSG TEXT=&apos.WRITE ERROR
                      RC=&&1&apos.,APPLID=CMS,  *
         SUB=(DEC,((9),4))
         B     DONE
READERR  EQU   *
         LR    R9,R15
         APPLMSG TEXT='READ ERROR
                      RC=&&1',APPLID=CMS, *
         SUB=(DEC,((9),4))
         B     DONE
* Data areas
BUFFERS  DS    0D          * addresses of buffers
         DC    A(OUTBUF1)
         DC    A(OUTBUF2)
         DC    A(OUTBUF3)
NUMBUFFS EQU   *-BUFFERS   * space needed to store buffer
                             addresses
LENGTHS  DS    0D          * addresses of buffer lengths
         DC    A(OUTBUFL1)
         DC    A(OUTBUFL2)
         DC    A(OUTBUFL3)
READBUF  DC    X'000000'   * need at least 1 byte for read
                             buffer
*
* Output buffers
OUTBUFS  DS    0D
*
* SAMPLE
OUTBUF1  DC    X'C21140401D60'
         DC    C'            SSSSS     AAAAA    MM   MM'
         DC    C'   PPPPPP    L         EEEEEEE            '
         DC    C'           S         A     A   M M M M'
         DC    C'   P     P   L         E                  '
         DC    C'            SSSSS    AAAAAAA   M  M  M'
         DC    C'   PPPPPP    L         EEEE               '
         DC    C'                 S   A     A   M     M'
         DC    C'   P         L         E                  '
         DC    C'            SSSSS    A     A   M     M'
         DC    C'   P         LLLLLLL   EEEEEEE            '
OUTBUFL1 EQU   *-OUTBUF1
*
* CONSOLE
OUTBUF2  DC    X'C211C7601D60'
         DC    C'       CCCCCC    OOOOO    NN    N    S'
         DC    C'SSSS     OOOOO    L         EEEEEEE       '
         DC    C'      C         O     O   N N   N   S '
         DC    C'        O     O   L         E             '
         DC    C'      C         O     O   N  N  N    S'
         DC    C'SSSS    O     O   L         EEEE          '
         DC    C'      C         O     O   N   N N     '
         DC    C'    S   O     O   L         E             '
         DC    C'       CCCCCC    OOOOO    N    NN    S'
         DC    C'SSSS     OOOOO    LLLLLLL   EEEEEEE       '
OUTBUFL2 EQU   *-OUTBUF2
*
* TEST
OUTBUF3  DC    X'C2114F401D60'
         DC    C'                     TTTTTTT   EEEEEEE'
         DC    C'    SSSSS    TTTTTTT                      '
         DC    C'                        T      E      '
         DC    C'   S            T                         '
         DC    C'                        T      EEEE   '
         DC    C'    SSSSS       T                         '
         DC    C'                        T      E      '
         DC    C'         S      T                         '
         DC    C'                         T      EEEEEEE'
         DC    C'    SSSSS       T                      '
OUTBUFL3 EQU   *-OUTBUF3
         REGEQU
         END
```

APPLMSG

Use the APPLMSG macro to retrieve a message from a message repository and, optionally, to display a message at your terminal.

APPLMSG is the recommended replacement for the LINEDIT macro in new programs and in programs being converted to exploit 31-bit addressing. APPLMSG contains most of the functions of the LINEDIT macro; in addition, you can use it to specify a message number rather than coding the entire message text. This allows for more flexibility in your programs because:

- It keeps your messages in a central location.
- You can use the same repository for several different programs.
- If you need to change the text of a message, you need to do it in only one place.
- If you need to support more than one language, you can use different repositories.
- You can create your own repository in any language.

For APPLMSG syntax information and examples, see *z/VM: CMS Macros and Functions Reference*.

Example of a Message Repository

The following example shows an external repository file, RUBUME REPOS, which is stored on a disk. You can view, edit, and update this message repository.

```
*
* This is an example of a message repository file made with XEDIT
* and maintains alignment of heading.
* You can code a file similar to this for your application.
*
& 3 Line specifies the substitution
character + no. of digits
02000101  |...+....1....+....2....+....3....+....4....+..
02000101  ..5....+....6....+....7....+....8
03000101  *                                         &2
03000101                   *
04000101  Header 1         &2
04000101  Header 2         &2
04000101  Header 3         &2
04000101  Header 4         &2
05000101  Smith            &2
05000101  Jones            &2
05000101  Johnson          &2
05000101  Parker           &2
05000201  Bill             &2
05000201  John             &2
05000201  Mark             &2
05000201  Joe              &2
```

Figure 15. Sample Repository - RUBUME REPOS

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

Line Numbers
  Explanation

1 - 5
  Comment lines.

6
  The control line.

  The ampersand (&) specifies the substitution character and the 3 specifies the number of message number digits to display (you should specify 3 or 4 for this value).

7 - 8
The first message is number 0200. This message has only one format and will be displayed as one line.

9 - 10
The second message is number 0300. This message has only one format and will be displayed as one line. The substitution character, &2, has a value of null and is used to maintain alignment.

11 - 14
The third message is number 0400. This message has only one format and will be displayed as one line. The substitution character, &2, has a value of null and is used to maintain alignment.

15 - 22
The fourth message is number 500. This message has two formats; either format will be displayed as one line. The substitution character, &2, has a value of null and maintains alignment.

For more information on message repositories, see the *z/VM: CMS Application Development Guide*.

Sample Program Using a Message Repository

Following is a sample program that uses the APPLMSG macro to access messages in RUBUME REPOS.

```
JEWEL    CSECT
         LR    R12,R15          Load base register
         USING JEWEL,R12
 *
 *
         LA    R5,DUMMY         Need a dummy substitution value
         APPLMSG APPLID=RUB,NUM=200,HEADER=NO
         APPLMSG APPLID=RUB,NUM=300,HEADER=NO,COMP=NO,
               SUB=(CHARA,((R5),1))
         APPLMSG APPLID=RUB,NUM=400,HEADER=NO,COMP=NO,
               SUB=(CHARA,((R5),1))
         APPLMSG APPLID=RUB,NUM=500,FMT=2,HEADER=NO,COMP=NO,
               SUB=(CHARA,((R5),1))
 *
 *
 FINI    EQU   *
         SR    R15,R15          Change for Return Code
         BR    R14              Return
         SPACE 1
         DS    0D
 DUMMY   DC    CL2'  '
         REGEQU
         END
```

Figure 16. Sample Program — JEWEL ASSEMBLE

Let's look at how APPLMSG is used in this program. In the fourth call to APPLMSG:

APPLID=RUB
Specifies RUBUME REPOS as the message repository that issues the messages.

NUM=500
Specifies message number 500 will be displayed.

FMT=2
Specifies format 2 of message 500 will be displayed.

HEADER=NO
Specifies no header will be created for the message.

COMP=NO
Specifies not to remove multiple blanks and the SUB parameter must be used in order for substitution indicators to be replaced. Therefore, the value stored at ‘DUMMY’ (blank) is substituted for all occurrences of &1 and any other substitution variables have a value of null.

SUB=(CHARA,((R5),1))

Specifies the value with a length of 1 stored at the address in register 5 will be substituted at every occurrence of &1. (&the specified substitution character in RUBUME repository.) Any subsequent substitution variable (for example, &2) will have a value of null.

For more information on substitution, see the *z/VM: CMS Application Development Guide*.

The output of this program will look like this:

```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
*
Header One      Header Two       Header Three     Header Four
Bill            John             Mark             Joe
```

LINERD and LINEWRT Macros

Use the LINERD and LINEWRT macro instructions to read and write lines of input and output from the terminal. LINERD and LINEWRT work from above or below 16MB and you can use them when CMS runs in full-screen mode or line mode. For more information, see *z/VM: CMS Macros and Functions Reference*.

Example

In the following example, LINERD reads one line of data and LINEWRT writes one line of data.

```
TESTWIR  CSECT
         STM   R14,R12,12(R13)
         BALR  R12,0
         USING *,R12
         ST    R13,SAVE13
         LA    R15,SAVEAREA
         ST    R15,8(R13)
         LR    R13,R15
* This program prompts users for their names and then greets
* them accordingly.  The name is invisible when typed in
* and an alarm rings when the greeting is complete.
* Depending on the display device, the greeting is typed
* in yellow.
*

         LINERD DATA=(BUFFER,12),PROMPT='WHO ARE YOU?',
         TYPE=INVISIBLE
         LINEWRT DATA=(JUNK,31),COLOR=YELLOW,ALARM=YES
         BR 14
         DS    0D
JUNK     DC    C'Hello '
BUFFER   DC    CL12' '
         DC    C', nice shoes.'
         REGEQU
SAVEAREA DC    18F'0'
SAVE13   EQU   SAVEAREA+4
         END   TESTWIR
```

Figure 17. A Sample Program That Reads and Writes One Line of Data

Reading and Writing Multiple Lines

When you have multiple reads and writes to perform, such as when using input panels, specify the FORM=MULTIPLE option with the LINERD and LINEWRT macros. This improves performance by reducing the number of SVC calls required to perform multiple I/O.

LINERD and LINEWRT use the same mechanism for handling multiple I/O. This mechanism is a **linked list** data structure composed of descriptors and text buffers. The actual structure of the linked list and the descriptors used by the LINERD macro is only slightly different from that used by the LINEWRT macro. This allows efficient updating of a virtual screen by using the input descriptors as output descriptors with the LINEWRT macro.

The first element of the linked list is a special descriptor (input for LINERD and output for LINEWRT) that only contains the cursor location (and the key pressed for LINERD) and a pointer to the next descriptor. The first descriptor can be used alone when:

- FORM=MULTIPLE on LINERD queries the cursor and key pressed.
- FORM=MULTIPLE on LINEWRT positions the cursor.

Each of the following descriptors contains information about the input or output text and a pointer to that text.

Because a typical use of multiple reads and writes might involve creating input panels, we will discuss how the LINEWRT macro performs multiple writes and then see how to do multiple reads with the LINERD macro.

Using the LINEWRT Macro for Multiple Outputs

Specify FORM=MULTIPLE on the LINEWRT macro to designate the multiple output format, which means that you will provide a chain of output descriptors. You can use the LWRD macro to define each output descriptor.

Use the DATA parameter to specify the address of the first output descriptor in the chain. The *length* field on DATA is ignored, if specified. You must specify only the cursor setting information in the first LWRD. Use the other LWRDs to completely describe each output to be displayed.

The LWRDs are chained together as follows:

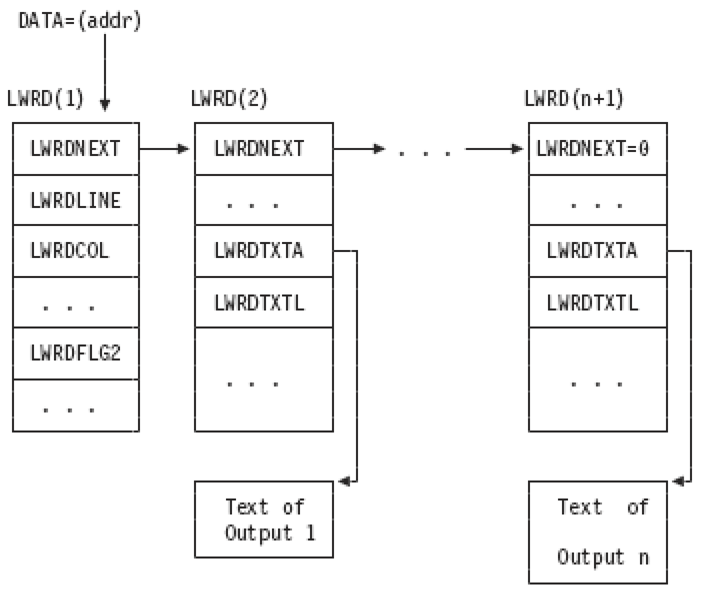


Figure 18. Chain of Output Descriptors

When FORM=MULTIPLE is specified, each output in the chain (including its text and attributes) is placed in the virtual screen output queue created by VSCREEN DEFINE. To move the text from the queue to the virtual screen, you can issue the VSCREEN WAITREAD, VSCREEN WAITT, or PSCREEN REFRESH command. This will move all the outputs in the queue to the virtual screen.

The LINERD macro can also be used to move the text from the output queue to the virtual screen when the read is satisfied from the terminal rather than the input queue or the stack. The LINEWRT macro works the same for a single output as it does for multiple outputs.

If you specify FORM=MULTIPLE and omit VNAME or specify VNAME=CMS when CMS is in line mode, then the outputs are written using WRTERM. This is the same way LINEWRT works for single outputs. Only the LWRDPRTY flag of the output descriptor is checked to determine whether this is a priority write. If you are writing the color (using the LWRDCLRF field), extended highlighting (using the LWRDEXHF field), or PSS buffer (using the LWRDPSSF field), or to the reserved area of the virtual screen, however, then the output is ignored in line mode and a WRTERM is not issued.

Using the LINERD Macro for Multiple Inputs

To request the multiple input format, specify FORM=MULTIPLE on the LINERD macro. Use the DATA parameter to specify the address and length of an application buffer to hold the chain of input descriptors (called LRDDs) returned by the LINERD macro.

The format of the information returned by LINERD in the application buffer is as follows:

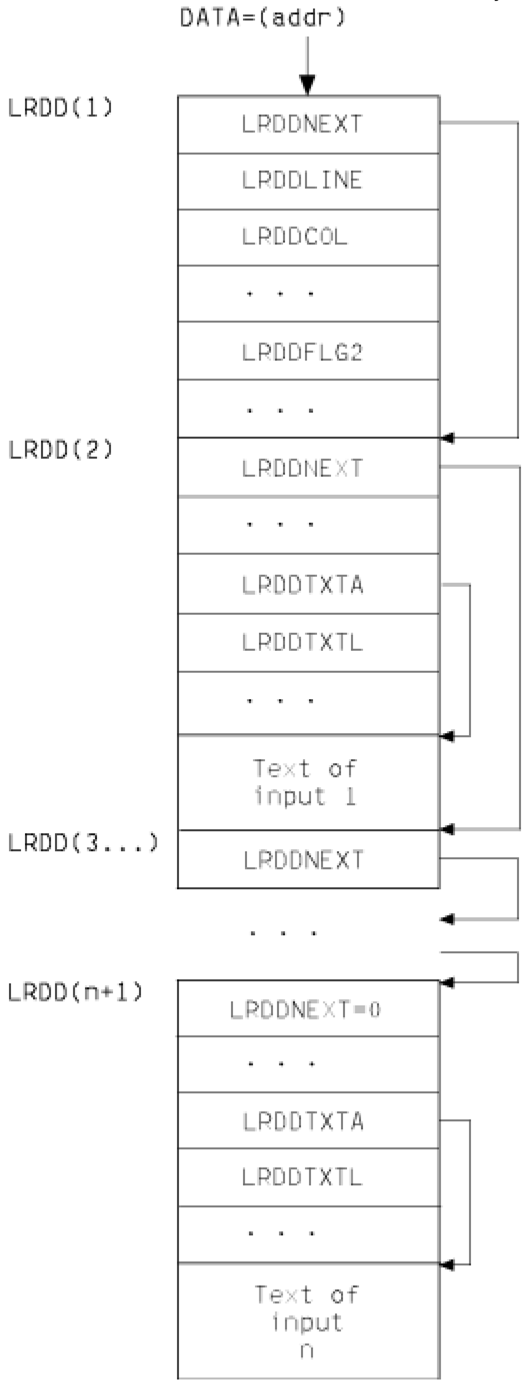


Figure 19. Format of Information Returned by LINERD Macro

The first LRDD in the chain returns the cursor position and the key pressed. The other LRDDs return the input attributes and the pointer field (LRDDTXTA) that contains the address of the input text read. This input text immediately follows the descriptor with which it is associated.

If the application buffer was not large enough to hold all of the inputs, a return code of 13 is returned to the application and the buffer length required for the next input is stored at the full-word specified by *addr2* of NUMRD.

There are two cases to consider when the buffer is not large enough:

1. The cursor and key descriptor would not fit in the buffer. In this case, *addr2* returns the length required for the cursor and key descriptor only.
2. The buffer was large enough for at least the cursor and key descriptor. The value returned in *addr2* is the length required for:
    - The cursor and key descriptor
    - The next LRDD
    - The input text associated with that LRDD.

If you specify FORM=MULTIPLE, but no fields have been modified, the cursor and key descriptor is always returned, and the full-word location specified by *addr1* of NUMRD has the value of 1. You can specify the addresses for NUMRD as assembler labels or general registers (2-12) enclosed in parentheses. The NUMRD parameter is optional, and if you do not specify *addr1* or *addr2* then the number of inputs read or the length of the next input is not returned.

The LINERD macro reads input from the line mode console, program stack, or full-screen console according to the option specified on the TYPE parameter and the full-screen/line mode environment. FORM=MULTIPLE is ignored and only one input is returned in the single input format when:

1. TYPE=STACK is specified (or defaulted to), and the read is satisfied by the program stack.
2. The virtual screen name (VNAME parameter) is not specified or VNAME=CMS and CMS is running in line mode, and the input is read from the console in a line mode environment.

You must specify FORM=MULTIPLE to receive multiple inputs. The value located at *addr1* of the NUMRD parameter is the number of inputs (including the cursor specified and key in LRDD) in the application buffer upon completion of the LINERD macro. If FORM=MULTIPLE is ignored or not specified, the input is returned at *addr* of the DATA parameter in single input format. A value of 0 at the address *addr1* of the NUMRD parameter upon completion of LINERD indicates that the input has been returned in single input format.

When FORM=MULTIPLE is specified, the DATA parameter must be specified, and the other parameters are optional, as is the case for a single input. The LINE, COL, and PAD parameters, if specified, are ignored when FORM=MULTIPLE is specified.

Descriptor Mappings

The mappings of the descriptors are provided here for easy reference when going through the sample program. A more complete explanation of the descriptor fields appears in the *z/VM: CMS Macros and Functions Reference* with each macro.

LWRD Mapping

You can map the output descriptors (including the cursor descriptor) by using the LWRD macro:

LINEWRT Descriptor Mapping

```
LWRD     DSECT
LWRDNEXT DS    A            Pointer to next
                            LINEWRT descriptor (LWRD)
         D S    CL8         Reserved
LWRDLINE DS     F           Line number
LWRDCOL  DS     F           Column number
LWRDTXTA DS     A           Text address - for output LWRD
LWRDTXTL DS     F           Length of text - for output LWRD
LWRDFLDL DS     F           Output length in vscreen -
                            for output LWRD
*
LWRDFLG1 DS     XL1         Flag byte #1 - for output LWRD
```

```
LWRDNTRF EQU    X'80'       X... .... No nulls translation
*        EQU    X'40'       .X.. .... Reserved
*        EQU    X'20'       ..X. .... Reserved
*        EQU    X'10'       ...X .... Reserved
LWRDCSEF EQU    X'08'       .... X... MIXED/SBCS attribute
                                      is specified
LWRDOUTF EQU    X'04'       .... .X.. Field outlining
                                      is specified
LWRDPRTY EQU    X'02'       .... ..X. Priority write
LWRDRESO EQU    X'01'       .... ...X Reserved area
                                      of vscreen
LWRDOUTL DS     XL1         Field outlining byte
                            - for output LWRD
LWRDCSET DS     XL1         MIXED/SBCS attribute
                            - for output LWRD
         DS    CL6          Reserved
LWRDATTR DS    XL1          Attribute byte
                            - for output LWRD
LWRDCOLR DS    XL1          Color byte
                            - for output LWRD
LWRDEXHI DS    XL1          Extended highlighting byte
                            - for output LWRD
LWRDPSS  DS    XL1          PSS byte
                            - for output LWRD
*
LWRDFLG2 DS    XL1          Flag byte #2
                            - for output LWRD
LWRDPSSF EQU   X'80'        X... .... PSS is specified
LWRDEXHF EQU   X'40'        .X.. .... Extended highlight
                                      is specified
LWRDCLRF EQU   X'20'        ..X. .... Color is specified
LWRDDATF EQU   X'10'        ...X .... Update data buffer
*
         EQU   X'08'        .... X... Reserved
LWRDCRSF EQU   X'04'        .... .X.. Position cursor
                                      within field
*    EQU   X'02'            .... ..X. Reserved
LWRDPADF EQU   X'01'        .... ...X Padding with blanks
*
ORG            LWRDFLG2     Redefine flag byte #2
                            - for cursor LWRD
LWRDLCUR DS    X
LWRDSETC EQU   X'02'        .... ..X. Position cursor
                                      using curs LWRD LWRDRESC
EQU            X'01'        .... ...X Reserved area of vscreen
*
LWRDTXT  DS    X            Text writes a field,data,
                            color,exthi,pss
**************************************************************
***      Valid text codes                                   **
**************************************************************
LWRDFLDV EQU   X'00'        Define a field with
                            default vscreen attr.
LWRDFLDD EQU   X'01'        Define a field and
                            use descriptor attr.
LWRDDATT EQU   X'02'        Text is data to write
                            in predefined field
LWRDCLRT EQU   X'03'        Text is color codes
LWRDEXHT EQU   X'04'        Text is extended
                            highlighting codes
LWRDPSST EQU   X'05'        Text is PSS codes
*
LWRDRC   DS    X            Individual return code
*************************************************************
***      Valid return codes                                **
*************************************************************
*
LWRDOK   EQU   0            Function executed successfully
LWRDINVP EQU   24           User did not specify descriptor
                            correctly
LWRDINVL EQU   32           Specified line/column is
                            outside vscreen
LWRDNOST EQU   104          Insufficient storage was
                            available
*
LWRDLEN  EQU   *-LWRD       Length of LWRD
LWRDDBSZ EQU  (LWRDLEN+7)/8 Length in doublewords
```

LRDD Mapping

You can map the input descriptors (including the cursor and key descriptor) by using the LRDD macro. See the *z/VM: CMS Macros and Functions Reference* for more information on the LRDD macro.

```
LRDD     DSECT
LRDDNEXT DS    A              Pointer to next LINERD
                              descriptor (LRDD)
         DS    CL8            Reserved
LRDDLINE DS    F              Line number
LRDDCOL  DS    F              Column number
LRDDTXTA DS    A              Text address - for input LRDD
LRDDTXTL DS    F              Length of text following
                              this LRDD - for input LRDD
*
         DS    CL4            Reserved
*
LRDDFLG1 DS    XL1            Flag byte #1 - for input LRDD
LRDDRESI EQU   X'01'          .... ...X Reserved area
                                        of vscreen
LRDDOUTL DS    XL1            Field outlining byte
LRDDCSET DS    XL1            MIXED/SBCS field attribute
         DS    CL6            Reserved
LRDDATTR DS    XL1            Attribute byte - for input LRDD
LRDDCOLR DS    XL1            Color byte - for input LRDD
LRDDEXHI DS    XL1            Extended highlighting byte
                              - for input LRDD
LRDDPSS  DS    XL1            PSS byte - for input LRDD
*
LRDDFLG2 DS    XL1            Flag byte #2 - for cursor LRDD
LRDDRESC EQU   X'01'          .... ...X Reserved area
                                        of vscreen
*
         DS    XL1            Reserved
*
LRDDKEY  DS    XL1            Holds key pressed
                              - for cursor LRDD
LRDDLEN  EQU   *-LRDD         Length of LRDD in bytes
LRDDDBSZ EQU ((LRDDLEN+7)/8)  Length of LRDD in doublewords
*
```

Figure 20. LINERD Descriptor Mapping

Example of Creating a Panel

As an example, let's create a simple input panel to prompt the user to change a file identifier, read any changes the user makes to the identifier, and then display the new identifier. The panel we have in mind looks something like this:

```
Fix filename or filetype and press ENTER:
SAMPLE
ASSEMBLE
```

After preparing the chain of output descriptors to be used by the macro, we can create this panel with just one LINEWRT macro call. We will need one descriptor to position the cursor after execution of the macro (this will be the first LWRD in the chain). Then we will need a LWRD for the prompt sentence, plus another LWRD to specify the colors for the prompt (because we like it colorful). Finally, we will use one LWRD for the file name and another for the file type (we will treat them as separate inputs for illustrating multiple reads as well). Refer to *z/VM: CMS Macros and Functions Reference* for the format of the descriptors and an explanation of their fields.

Now that we know what we want the input panel to look like, we can set up the data area to simplify the coding. The data area (at the end of the program) should include the following fields:

```
*
* Data area.
*
DS    0D
OUTBUFF  DS    CL400 output buffer
INBUFF   DS    CL400 input buffer
FW1      DC    F'1'  fullword 1
FW2      DC    F'2'  fullword 2
```

```
FW3      DC    F'3'  fullword 3
FW4      DC    F'4'  fullword 4
FW5      DC    F'5'  fullword 5
FW8      DC    F'8'  fullword 8
FW9      DC    F'9'  fullword 9
FW39     DC    F'39' fullword 39
FW40     DC    F'40' fullword 40
NULL     DC    X'00' null character
BLANK    DC    X'40' blank character
A0       DC    A(0)  address 0
OUTLNG   DC    X'00' no outlining (default)
SBCS     DC    X'01' specify SBCS char set
PRNH     DC    X'30' protect/nohigh attribute
NPNH     DC    X'00' noprotect/nohigh attribute
BLUE     DC    X'F1' blue color
TURQ     DC    X'F5' turquoise color
NONE     DC    X'00' no exthi
NUMFLDS  DS    F     number of modified fields
TEXT2    DC    C'Fix filename or filetype and
                 press ENTER:'  LWRD2 text
TEXT3    DC    C'44442222222224444222222222444444444333334'
                 LWRD3 text
DEFNAME DC    CL8'SAMPLE'    default filename
DEFTYPE DC    CL8'ASSEMBLE'  default filetype
FILEID  DS    CL20           fileid
        ORG   FILEID
NAME    DS    CL8            filename
        DC C' '              blank
TYPE    DS    CL8            filetype
        DC    C' '           blank
MODE    DC    CL2'*'         filemode
HEADER  DC    C'The new fileid is:'  header
                for writing results
LRDD                         LINERD descriptor DSECT
LWRD                         LINEWRT descriptor DSECT
REGEQU                       register equates
END   EXAMPLE
```

The first several fields are self explanatory. Some of the other fields are explained in the following list:

- The OUTLNG and CHARSET (in the example, SBCS) fields, defined about halfway down in the listing, set the values in LWRDOUTL and LWRDCSET to indicate no field outlining and a single-byte character set.
- The PRNH and NPNH fields are used in the LWRDATTR field of the LWRDs.
- The BLUE and TURQ fields specifies color codes in the LWRDCOLR field.
- NONE is used in the LWRDEXHI field to specify that extended highlighting is not to be used when writing the prompt. (The attribute, color, extended highlighting, and other codes are listed and described in the *IBM 3270 Information Display System Data Stream Programmer's Reference**.)
- NUMFLDS will be used to keep track of the number of modified fields read (we will use it as the fullword of storage for the NUMRD parameter).
- TEXT2 specifies the text of the prompt to be written.
- TEXT3 specifies the colors to be used to write the prompt.

The rest of the fields are self-explanatory. Note that the LWRD and LRDD mapping macros are at the end of the data area.

Next, we can set up the program:

```
EXAMPLE  CSECT
         LR    R12,R15             load base register
         USING EXAMPLE,R12
         LA    R2,OUTBUFF          point to output buffer
         LA    R3,L'OUTBUFF        get length of buffer
         ICM   R3,B'1000',NULL     set pad character to null
         SR    R4,R4               set source address to zero
         SR    R5,R5               set source length to zero
         MVCL  R2,R4               clear out output buffer
```

Now, we can fill in the required fields of the cursor LWRD:

```
*
* Fill in the cursor LWRD (LWRD1).
*
```

```
       LA    R2,OUTBUFF          point to output buffer
       USING LWRD,R2
       LA    R3,LWRDLEN(R2)      calculate address of next LWRD
       ST    R3,LWRDNEXT         fill in pointer to next LWRD
       MVC   LWRDLINE,FW4        fill in line number
       MVC   LWRDCOL,FW2         fill in column number
       MVC   LWRDFLG2,NULL       clear flag byte 2
       OI    LWRDFLG2,LWRDSETC   set set-cursor flag
```

This code calculates and stores the address of the next LWRD and specifies where the cursor is to be positioned upon completion of the LINEWRT macro. Note that the LWRDSETC flag of the LWRDFLG2 field must be set to indicate that the information in this LWRD should be used to set the cursor. If the flag is not set, the cursor will be positioned according to the LWRDCRSF flag of LWRD5 (the last descriptor in the chain defining a field in the data area).

Next, we prepare the output descriptors that provide all of the information necessary for writing the outputs to the vscreen. The following code describes the prompt to be written:

```
*
* Fill in LWRD2, which writes a prompt.
*
     LR    R2,R3               point to this LWRD
     LA    R3,LWRDLEN(R2)      calculate address of next LWRD
     ST    R3,LWRDNEXT         fill in pointer to next LWRD
     MVC   LWRDLINE,FW3        fill in line number
     MVC   LWRDCOL,FW1         fill in column number
     LA    R4,TEXT2            get address of text
     ST    R4,LWRDTXTA         fill in text address
     LA    R4,L'TEXT2          get length of text
     ST    R4,LWRDTXTL         fill in text length
     MVC   LWRDFLDL,FW40       fill in field length
     MVC   LWRDFLG1,NULL       clear flag byte 1
     MVC   LWRDOUTL,OUTLNG     specify no outlining
     MVC   LWRDCSET,SBCS       specify SBCS
     MVC   LWRDATTR,PRNH       fill in attribute byte
     MVC   LWRDCOLR,BLUE       fill in color byte
     MVC   LWRDEXHI,NONE       fill in exhi byte
     MVC   LWRDFLG2,NULL       clear flag byte 2
     OI    LWRDFLG2,LWRDEXHF   set exthi-specified flag
     OI    LWRDFLG2,LWRDCLRF   set color-specified flag
     OI    LWRDFLG2,LWRDPADF   set pad-with-blanks flag
     MVI   LWRDTXT,LWRDFLDD    choose "define field" text code
     MVC   LWRDRC,NULL         clear LWRD return code
```

A few of the fields warrant a fuller explanation. Approximately in the middle of the code:

- We put a NULL value (X'00') in the LWRDFLG1 field (leaving the flags in this field *off*) because this is not a priority write, it is to be written to the scrollable area of the virtual screen, and we want nulls translation done.
- The LWRDOUTL field specifies that the input field will not be outlined.
- The LWRDCSET field is set to specify single-byte character set.
- The LWRDATTR field is set to a protected, no highlight field so the prompt cannot be modified by the user.
- Next, we set the LWRDCOLR field for blue color.
- In the LWRDEXHI field we specify not to have extended highlighting.
- Setting the LWRDEXHF flag indicates that the value in the LWRDEXHI field determines the extended highlighting (in this case, none).
- Setting the LWRDCLRF flag indicates that the color specified in the LWRDCOLR field is used when writing the prompt.
- Setting the LWRDPADF flag indicates that the data buffer is to be padded with blanks (instead of nulls) if the output field is longer than the actual text to be written.
- Setting the LWRDTXT field to LWRDFLDD (X'01') indicates that we are defining a field and that we are using the LWRDATTR field to specify the field attribute rather than using the default vscreen field attribute.
- Finally, we clear the LWRDRC field so we can be certain about any code returned.

The next descriptor specifies the colors to be used to write the prompt. Most of the code is basically the same as that in the preceding section :

```
*
* Fill in LWRD3, which writes various colors for the prompt.  (The
* text starts in column 2 with a length of 39 since the field is
* already defined.)
*
     LR    R2,R3              point to this LWRD
     LA    R3,LWRDLEN(R2)     calculate address of next LWRD
     ST    R3,LWRDNEXT        fill in pointer to next LWRD
     MVC   LWRDLINE,FW3       fill in line number
     MVC   LWRDCOL,FW2        fill in column number
     LA    R4,TEXT3           get address of text
     ST    R4,LWRDTXTA        fill in text address
     LA    R4,L'TEXT3         get length of text
     ST    R4,LWRDTXTL        fill in text length
     MVC   LWRDFLDL,FW39      fill in field length
     MVC   LWRDFLG1,NULL      clear flag byte 1
     MVC   LWRDFLG2,NULL      clear flag byte 2
     MVI   LWRDTXT,LWRDCLRT   choose "write color" text code
     MVC   LWRDRC,NULL        clear LWRD return code
```

As noted in the comment preceding the code, we specify column 2, because the start field was placed in column 1 when the field was defined. We also need to clear both flag bytes of this descriptor and specify the LWRDCLRT code for the LWRDTXT field. This indicates that this descriptor specifies the colors to be used for writing the output.

The next two sections of code fill in two more LWRDs.

```
*
* Fill in LWRD4, which writes the filename.
*
     LR    R2,R3              point to this LWRD
     LA    R3,LWRDLEN(R2)     calculate address of next LWRD
     ST    R3,LWRDNEXT        fill in pointer to next LWRD
     MVC   LWRDLINE,FW4       fill in line number
     MVC   LWRDCOL,FW1        fill in column number
     LA    R4,DEFNAME         get address of text
     ST    R4,LWRDTXTA        fill in text address
     LA    R4,L'DEFNAME       get length of text
     ST    R4,LWRDTXTL        fill in text length
     MVC   LWRDFLDL,FW9       fill in field length
     MVC   LWRDFLG1,NULL      clear flag byte 1
     MVC   LWRDOUTL,OUTLNG    specify no outlining
     MVC   LWRDCSET,SBCS      specify SBCS
     MVC   LWRDATTR,NPNH      fill in attribute byte
     MVC   LWRDCOLR,TURQ      fill in color byte
     MVC   LWRDFLG2,NULL      clear flag byte 2
     OI    LWRDFLG2,LWRDCLRF  set color-specified flag
     MVI   LWRDTXT,LWRDFLDD   choose "define field" text code
     MVC   LWRDRC,NULL        clear LWRD return code
*
* Fill in LWRD5, which writes the filetype.
*
     LR    R2,R3              point to this LWRD
     MVC   LWRDNEXT,A0        fill in pointer to next LWRD
     MVC   LWRDLINE,FW5       fill in line number
     MVC   LWRDCOL,FW1        fill in column number
     LA    R4,DEFTYPE         get address of text
     ST    R4,LWRDTXTA        fill in text address
     LA    R4,L'DEFTYPE       get length of text
     ST    R4,LWRDTXTL        fill in text length
     MVC   LWRDFLDL,FW9       fill in field length
     MVC   LWRDFLG1,NULL      clear flag byte 1
     MVC   LWRDOUTL,OUTLNG    specify no outlining
     MVC   LWRDCSET,SBCS      specify SBCS
     MVC   LWRDATTR,NPNH      fill in attribute byte
     MVC   LWRDCOLR,TURQ      fill in color byte
     MVC   LWRDFLG2,NULL      clear flag byte 2
     OI    LWRDFLG2,LWRDCLRF  set color-specified flag
     MVI   LWRDTXT,LWRDFLDD   choose "define field" text code
     MVC   LWRDRC,NULL        clear LWRD return code
```

At last, we are ready to display the input panel and read any changes the user makes to the file name and file type:

```
*
* Write the information and read any changes.
*
     LINEWRT DATA=(OUTBUFF),FORM=MULTIPLE  write the panel
     LINERD DATA=(INBUFF,400),FORM=MULTIPLE,WAIT=YES,       X
           NUMRD=(NUMFLDS)  read the user's changes
```

The only parameters we need to specify on the LINEWRT macro are DATA (with a pointer to the first LWRD in the chain) and FORM=MULTIPLE. All the other information required for the writes is specified in the LWRDs.

To use the LINERD macro for multiple inputs, we need to provide a buffer for the input descriptors (LRDDs) containing the data read. In addition, we specify NUMRD so we can easily tell how many inputs are being returned in the buffer. NUMFLDS designates the fullword of storage for the NUMRD value. If NUMFLDS contains a 0 upon completion of LINERD, the input is returned in single input format. Otherwise, the inputs are returned in multiple input format. Recall that the number returned in NUMFLDS indicates the number of modified fields read plus one for the cursor and key descriptor. If no fields were modified, NUMFLDS contains a 1, and the multiple input format is used to return the cursor and key descriptor in the buffer. The WAIT=YES option just specifies the status area message to indicate that the program is waiting for a response from the user.

After the LINERD macro has executed, we need to determine what was read. The following section of code checks the value in NUMFLDS to see whether more than the cursor LRDD is being returned, in which case, control will pass to a loop that can process the LRDDs in the buffer.

```
*
* Fill in NAME and TYPE from the user's response.
*
     MVC   NAME,DEFNAME       start with default name
     MVC   TYPE,DEFTYPE       start with default type
     CLC   NUMFLDS,FW1        is cursor LRDD the only one?
     BE    RESULTS            yes, go write the results
     LA    R2,INBUFF          point to LRDD1 (cursor/key LRDD)
     USING LRDD,R2
```

In the following code, note that we specify that the LRDD mapping is to be used for the fields used in the following code. The loop code checks each input read, taking only those fields that have been modified. To keep the example short, we use LINEWRT in the default (SINGLE) form to display the results.

```
USING LRDD,R2
LOOPTOP  DS    0H
         L     R2,LRDDNEXT        move to next LRDD
         CLC   LRDDLINE,FW4       is this a modified filename?
         BNE   CHKTYPE            no, go check if it's filetype
         L     R6,LRDDTXTA        point to text
         L     R7,LRDDTXTL        get length of text
         ICM   R7,B'1000',BLANK   set pad character to blank
         LA    R8,NAME            point to filename
         LA    R9,L'NAME          get length of filename
         MVCL  R8,R6              move response into NAME
         B     LOOPEND            go to end of loop
CHKTYPE  DS    0H
         CLC   LRDDLINE,FW5       is this a modified filetype?
         BNE   LOOPEND            no, go to end of loop
         L     R6,LRDDTXTA        point to text
         L     R7,LRDDTXTL        get length of text
         ICM   R7,B'1000',BLANK   insert pad character
         LA    R8,TYPE            point to filetype
         LA    R9,L'TYPE          get length of filetype
         MVCL  R8,R6              move response into TYPE

LOOPEND  DS    0H
         CLC   LRDDNEXT,A0        is pointer zero?
         BNE   LOOPTOP            no, go through loop again
*
* Write the results and exit.
*
RESULTS  DS    0H
         LINEWRT DATA=(HEADER),COLOR=BLUE  write a heading
```

```
          LINEWRT DATA=(FILEID),COLOR=BLUE  write the new fileid
          BR    R14                return to caller
```

Before running this program, we must SET CHARMODE ON so we will be able to see the character attributes we defined in LWRD3.

After execution of the program, the screen looks something like this:

```
Fix filename or filetype and press ENTER:
SAMPLE
ASSEMBLE
The new fileid is:
RAZZLE   DAZZLE   *
```

Considerations for Writing Applications in Full-Screen CMS

CMS provides a windowing capability called full-screen CMS that allows you to manage several pieces of information on the physical screen at the same time. The following considerations apply when writing applications that will execute when full-screen CMS is active.

- If the CP SLEEP command is entered while full-screen CMS is on, it will appear that your terminal is hung. Any write to the terminal will unlock the keyboard. (See the CP SLEEP command in the *z/VM: CP Commands and Utilities Reference* for more information.)
- Some programs embed the hexadecimal code X'1D' to affect the highlighting and color attributes of output. In full-screen CMS,X'1D' is a nondisplayable character and does not affect the attributes of data following it. The VSCREEN DEFINE, SET VSCREEN, and VSCREEN WRITE commands (as well as the LINEWRT macro) let programs specify attributes for data in full-screen CMS.
- Error messages generated from windowing commands are displayed based on how the command was executed:
  - When called from a program or from an exec procedure with &CONTROL NOMSG, then no message is displayed and only the return code is set.
  - When called from an exec procedure with ADDRESS COMMAND, then the variable *message.1* is set to the message text and *message.0* is set to 1 to indicate the number of message variables set.
  - In other cases, such as a command entered from the command line or from an EXEC procedure with ADDRESS CMS, the message is displayed at the terminal. These messages are edited based on the CP EMSG setting.
- When returning to full-screen CMS from an application that performs its own full-screen management, your screen can contain mixed data. Press the CLEAR key to scroll forward and refresh the screen. Alternatively, you can enter the command SET FULLSCREEN SUSPEND before executing the application.
- The following messages are not trapped by the IUCV Message All System Service and are directly sent to the terminal:
  - Asynchronous CPCONIO (including PER/TRACE events)
  - EMSGs not generated as part of a DIAG X'08' instruction
  - Accounting messages.
- Certain applications must be changed to correctly work in the full-screen CMS environment:
  - Programs which issue a 3215 SIO to do a line-mode read
  - Programs which issue DIAGNOSE code X'58' to do full-screen I/O.

  Line-mode output and prompts written by these applications will not be immediately displayed in the full-screen CMS environment. Also, messages and warnings from other computer users are not displayed until you exit these applications. These applications should be changed to use CMS macros:

  - Change 3215 SIO to use LINERD macro
  - Change DIAGNOSE code X'58' to use CONSOLE macro

Alternatively, to avoid problems viewing output during the execution of such programs, you can temporarily suspend full-screen CMS (SET FULLSCREEN SUSPEND) before running the application and then resume your session (SET FULLSCREEN RESUME) upon exiting the application.

- When full-screen CMS is on, most CMS console output is not passed to CP. In addition, applications that use the IUCV Message System Service (*MSG) and SET VMCONIO IUCV will not trap all CMS output. Before running such applications, it is recommended to suspend full-screen CMS.
- If an application runs disconnected using the Single Console Image Facility (SCIF) to communicate with the disconnected user, the primary user must have FULLSCREEN set OFF or SUSPEND. If FULLSCREEN is ON, message HCPSEC068E will be received instead of the normal response following the second or third attempt to SEND a command to the primary machine.
- Full-screen CMS initialization issues the CP TERMINAL BRKKEY NONE command. Application developers may want to reset the CP TERMINAL BRKKEY to PA1 when debugging.
- You can specify the EXIT parameter for the OPEN function of the CONSOLE macroinstruction to handle unrequested device interrupts.
- If EXIT is specified, **do not** define an interruption routine using the HNDINT macro for the same device. Use of the CONSOLE and HNDINT macros is mutually exclusive. CONSOLE OPEN with EXIT supersedes an HNDINT routine when the interrupt is requested. Therefore, if you want to do I/O to a 3270 device, use the CONSOLE macro instead of the HNDINT macro.
- When using full-screen CMS, an application which uses the OS/MVS STAX macro or the HNDINT macro may work differently than when not using full-screen CMS. In line-mode CMS, only the attention interrupts caused by the ENTER key are passed to the OS/MVS STAX or HNDINT exit. However, in full-screen CMS, attention interrupts caused by any key may be passed to the exit. If your exit is not prepared to handle interrupts other than those caused by the ENTER key, then full-screen CMS should be suspended.
- While in FULLSCREEN CMS, if an application issues a CONSOLE WRITE to a dialed device when there has been linemode output to the virtual console, the WM window will not be popped to refresh the screen. Therefore, the linemode output will not be displayed until the application terminates.

Chapter 9. CMS File System

This chapter discusses the following topics:

- What are CMS files and how are they maintained by CMS
- What are byte file system (BFS) files and how are they maintained by CMS
- How can applications interface to CMS files and BFS files to perform file input and output
- Using XEDIT to Access Files in Storage.

Overview of the CMS File System

This section gives a high level overview of how CMS manages files. Reviewing these concepts should help you design your applications to make the best use of CMS files and BFS files. This section discusses the attributes of CMS files and BFS files, the information CMS maintains about CMS files and BFS files, and the operations you can perform on CMS files and BFS files.

What Is a CMS File?

If we start from the general user's point of view, we know a virtual disk, or minidisk, is a place where we can collect files. Files are what we use to collect logically related data or records. CMS manages the data in files and the files placed on disks using a mapping system. This mapping system is a tree-like structure of pointers and data, where pointers serve as indexes to pieces of data. The amount of pointers and data possible is based on the physical DASD block size of the CMS disk.

CMS disks are formatted into blocks that can be 512, 1K-, 2K-, or 4K bytes. The block size used is determined when a minidisk, or virtual disk, is formatted. Thus, one disk does not contain a mixture of block sizes. A file consists of data blocks and pointer blocks, which are the same size. The data in a file is broken up into fixed size portions, which are stored on data blocks. Pointer blocks chain the data blocks together. Pointer blocks either point to data blocks or to other pointer blocks.

Choosing an appropriate block size to format a disk depends on its intended use. A 4K block size will optimize the I/O if the disk is to contain large files with no missing records (dense). A block size of 1K is more appropriate when creating many small files or sparse files. For example, PL/I regional files are sparse and they may allocate more space on a 4K disk than on a 1K disk; therefore, the smaller block size is preferable.

The larger the block size of the disk, the greater the amount of storage required for input/output buffers. Each buffer that the system needs must be a contiguous block of system keyed storage. The size of this area of storage is the block size of the disk.

Other Architectures

The types of disks and files just discussed (based on 512, 1K, 2K or 4K block sizes) are part of the Enhanced Disk Format (EDF) architecture. CMS also supports Shared File System (SFS) architecture which is used for files residing in a CMS file pool.

SFS files are represented in storage in the same way EDF files are. The significant difference is that SFS files are always formatted with 4K-byte blocks. SFS shares CMS programs and data among users. SFS supports hierarchical directories and file sharing. Data kept by a file server virtual machine is on server-owned disk space, which is shared by all owners of files in that file pool. Requests for data from a CMS user machine are sent across an APPC/VM link to the server machine. Requests can be in the form of commands, CSL routine calls, or macros.

What is a BFS File?

OpenExtensions is the implementation of IEEE POSIX standards for system interfaces and threads on z/VM. Included in OpenExtensions is a POSIX-compliant file system called the Byte File System (BFS). BFS is a companion to the SFS that provides a byte-stream view of files. That is, BFS files consist of continuous streams of individual bytes of data. Such files have no record format or other record file attributes. The interpretation of BFS files is defined by the applications that use them. For example, the byte stream may include special characters that control the interpretation of the file.

CMS supports the generation of byte file systems as file spaces in CMS file pools. Multiple byte file systems can be enrolled in the same file pool, and byte file systems can reside in the same file pool as SFS file spaces. The primary programming interface for manipulating BFS files is the set of OpenExtensions CSL routines documented in the *z/VM: OpenExtensions Callable Services Reference*. However, CMS file pools can support both BFS data and SFS data with common administration tasks and system-managed storage. To do this, CMS gives BFS files the *appearance* of having some CMS file attributes.

What File Information Does CMS Maintain?

Associated with each CMS disk is a file directory, which contains an entry for every CMS file on the disk. When you access a disk or SFS directory, a file directory is stored in your virtual machine. The entries in the file directory for each CMS file are called File Status Tables (FST). The FST describes the attributes of the file. Attributes of a file include:

- File name
- File type
- File mode
- Record format
- Logical record length
- Number of records in the file
- File origin pointer
- Number of data blocks
- Number of pointer block levels
- Date and time of last update.

In addition, CMS maintains *extended file attributes*:

- File system type (SFS or BFS)
- Recoverability
- Overwrite
- Date of Last Reference
- Creation Date and Time
- Migration indicator
- DFSMS/VM* Related attributes
- Maximum blocks used for the file
- System blocks used for the file
- Date and time of last change

for files stored in file pools. These extended file attributes, described in the following section, are not maintained in the FSTs. Your programs can retrieve them by using callable services library (CSL) routines.

BFS files have two sets of file attributes associated with them. The first set consists of the path name, size in bytes, POSIX user ID (UID) and group ID (GID), permission bits, and other attributes associated with the file in accordance with the IEEE POSIX 1003.1 standard. For a full description, see the *z/VM: OpenExtensions User's Guide*. Because BFS files are stored in CMS file pools, they also have a set of CMS record file system attributes associated with them. For example, BFS files have a system-generated CMS file name and file type, and are represented to the record file system CSL routines that manipulate them as fixed-length record-format files with a logical record length of 1.

File Attributes

File Name, File Type, File Mode

When you create a file in CMS, you name it using a file identifier. The file identifier consists of three fields: file name, file type, and file mode (or directory name for SFS files). The file name and file type can each be from one to eight characters. Valid characters are A-Z, a-z, 0-9, #, @, +, - (hyphen), : (colon), and _ (underscore). The file mode indicates the file mode letter (A-Z) currently assigned to the SFS directory or minidisk where you want the file to reside. See the *z/VM: CMS User's Guide* for a discussion of valid file names, file types and file modes.

Every CMS file, regardless of whether it resides in a file space or on a minidisk, has a file mode number associated with it. The file mode number is established when the file is created. Some file mode numbers have special meanings:

File Mode Number 0

For the minidisk environment, file mode number 0 is used to make files private. For the SFS environment, file mode number 0 means the same as file mode number 1.

File Mode Number 1

Used for reading and writing files. It is the default file mode number.

File Mode Number 2

Used for reading and writing files. Also used to denote subsets of files.

File Mode Number 3

Files are erased after they are read. You can use file mode number 3 if you do not want to maintain copies on your minidisks or in your file space.

File Mode Number 4

Files are in OS simulated data set format. These files are created by OS macros in programs running in CMS.

File Mode Number 5

Used for reading and writing files. Also used to denote subsets of files.

File Mode Number 6

For EDF files, indicates that the *update-in-place* attribute of a CMS file is in effect. This means that the existing records of a file are written back to their previous location on the minidisk, rather than in a new slot.

WARNING: When modifying an existing file mode number 6 file, it is possible to damage the file, or even the entire minidisk on which it resides. This damage occurs when some of the updates made to the file or disk by an application are updated in place, but CMS terminates (requiring a re-IPL of CMS) before it can write all of the data to disk.

For SFS files, this file mode number is treated the same as file mode 1.

Note: If you want your files to have the *update-in-place* attribute, (where updates to physical blocks of data are made directly in that block), see the *z/VM: CMS Application Development Guide* for a description of the methods you could use to specify this attribute.

File Mode Numbers 7-9

Reserved for IBM use.

A BFS file is identified in CMS by a system-generated numeric file name and file type. (The BFS file name, which is the last component of the path name that identifies the file in the OpenExtensions interface, cannot be used in the CMS record file interface.) This CMS file ID is guaranteed to be unique within a BFS file space. You can obtain the CMS file IDs for BFS files by using DMSOPDIR CSL routine with an intent of FILEEXT or by using the OPENVM LISTFILE command with the NAMES option.

The CMS file mode number of the BFS file, when it applies, is always 1.

Record Formats

From the user's point of view, a file consists of from one to 2,147,483,647 (2^{31}-1) records, each of which consists of from one to 2^{31}-1 bytes of data (a record in a file with variable-length records is further restricted to 65,535 bytes of data). This limit is not normally significant. The amount of space available on the storage medium will usually be the significant limit.

When viewed through the CMS record file system interface, each "record" of a BFS file consists of a single byte. A BFS file may contain more than 2^{31}-1 records (bytes). However, the CMS record file interface cannot handle a file that large.

A file has one of two record formats:

F-Format

When all records in a file must have the same length, the file is said to be an F-format file and its records are said to be fixed-length records

V-Format

When the records in a file may have different lengths, the file is said to be a V-format file and its records are said to be variable-length records.

The record format of a file is determined when the file is created and cannot be changed thereafter. For an existing file, the record format is taken from the file's attribute information. For a new file, the record format is determined from the parameters specified on the macros or routines that will open the file.

BFS files are always F-format.

Logical Record Length

The length of each record in a new F-format file is the length of the first record written to the file — this length cannot be changed by any subsequent write to the file. The logical record length of a new empty file is determined by the LRECL value specified on the command or CSL routine that creates it. Empty files created from non-empty files may still retain their old values. The length of each record in a V-format file is recorded by a two-byte length prefix stored immediately before the record itself. A record must not be null, that is, have a length of zero. The length of the longest record in the file is stored in the file's directory entry (in the logical record length field of the FST) when the file is closed. If the file is written to later, a longer record may be appended to the file, in which case the length stored in the file's directory entry will be updated when the file is closed.

For BFS files, the logical record length is always 1.

File Origin Pointer, Number of Data Blocks and Pointer Levels

The File Origin Pointer (FOP) identifies the highest level pointer block or data block. The pointer blocks locate the next lower level of pointer blocks or the data blocks that contain the actual data for the file. Pointer blocks can go as high as 6 levels (or 6 levels deep on a tree), where level 1 pointer blocks point directly to the data blocks. The highest number of data blocks per file possible is 2^{31}-1. The number of data blocks depends on how the disk is formatted, on the size of the file, and if the file has sparse blocks.

For BFS files, CMS simulates the pointer blocks. The number of data blocks is equal to the number of bytes in the file divided by the block size (4096 bytes).

Record Number and Number of Records

Each record in a file is assigned a number known as its record number or its position number. The first record in a file has a record number of one and each succeeding record has a record number one greater than the preceding record. Thus, the number of records in a file is the greatest number of any record written.

A minidisk file cannot have zero records. SFS files can have zero records; empty files can be created using the CREATE FILE or XEDIT commands, DMSCRFIL CSL routine, or by specifying the ALLOWEMPTY parameter on the DMSOPEN CSL routine. You can also make an SFS file empty by using the ERASE command or DMSERASE CSL routine with the DATAONLY option.

For BFS files, the record number attribute has the same interpretation as for CMS record files. Because each record in a BFS file consists of a single byte, the number of records in a BFS file is equal to the size of the file in bytes. A BFS file may be empty.

Date and Time of Last Update

The date and time is stored in 6 bytes (yy mm dd hh mm ss), where each byte holds two decimal digits. In a flag byte is a bit to indicate the century. A setting of '0' indicates the time frame of 1900 to 1999, a setting of '1' indicates the time frame of 2000 to 2099. This is the date and time that the accessed file was last updated. Some CMS commands, such as COPYFILE, that transfer data from one location to another could copy over the date and time of the existing file. In that case, the date and time would not actually reflect the last time data was written to the file.

For BFS files, these attributes have the same interpretation in CMS as for CMS record files. However, when stored in the catalogs these two attributes are translated into a single POSIX attribute called MTIME that consists of the total number of seconds from January 1, 1970.

Recoverability

The recoverability attribute specifies whether the file is to be recoverable or irrecoverable. This attribute applies only to files stored in CMS file pools.

When a file is recoverable, uncommitted changes are backed out as the result of an application initiated rollback. Files are recoverable unless specified otherwise using a CSL routine or the FILEATTR command.

Irrecoverable files are not rolled back if an application initiated the rollback. As many updates as possible will be committed.

BFS files are recoverable when using the CMS record file interface.

For more information on committing and rolling back changes, see the *z/VM: CMS Application Development Guide*.

Overwrite

The overwrite attribute specifies whether updates to a file will be made in place. This attribute applies only to files stored in CMS file pools.

Most files are not updated in place, unless you specify otherwise using a CSL routine or the FILEATTR command. These files are to be shadowed when written so that readers see a consistent version of the file from open to close.

With update-in-place files, updates are to be made in place where possible. Users that have an INPLACE file open for read would have to re-open the file to see extensions (new blocks or records) that have been written and committed to the file.

BFS files are always NOTINPLACE.

Date of Last Reference

The date of last reference is the date on which the file was last read or updated. If the file hasn't been read or updated since it was created, the date of last reference is the date of file creation.

The difference between the date of last reference and the date attribute (described above) is that the date of last reference is updated when a file is read—the date attribute is not.

CMS maintains the date of last reference only for files that reside in CMS file pools. It is stored in the file pool catalogs—not in the FSTs as basic file attributes are.

The date of last reference is based on Coordinated Universal Time (UTC) at the time of the reference. The date attribute, on the other hand, is based on the local time. This can cause discrepancies between the attributes, depending on the geographic location of your processor. This difference is important to remember when you are coding an application that uses the date of last reference. You might, for example, want to convert the local date to the UTC date.

For BFS files, this attribute has the same interpretation in CMS as for CMS record files. However, when stored in the catalogs this attribute is translated into a POSIX attribute called ATIME that consists of the total number of seconds from January 1, 1970.

You can use CSL routines (such as DMSEXIFI and DMSGETDX) to retrieve the date of last reference for a file. You can also use CSL routines (DMSFILEC, DMSOPEN, and DMSOPBLK) to inhibit the updating of the date of last reference. The date of last reference is intended for use by application programs. It is displayed by the FILELIST and LISTFILE commands through the use of the ALLDATES option.

For more information on how to use CSL routines, see the *z/VM: CMS Application Development Guide*.

Creation Date

The creation date is the date the file was created. It is based on Coordinated Universal Time.

CMS maintains the creation date only for files that reside in CMS file pools. It is stored in the file pool catalogs—not in the FSTs as basic file attributes are.

You can specify a creation date of your own choosing when you use CSL routines to create a file. Once the file is created, you cannot change the creation date.

For BFS files, this attribute has the same interpretation as for CMS record files. However, there is no equivalent POSIX attribute.

You can use CSL routines such as DMSEXIFI (Exist - File) and DMSGETDX (Get Directory - Extended) to retrieve the creation date for a file. The creation date is intended for use by application programs. It is displayed by the FILELIST and LISTFILE commands through the use of the ALLDATES option.

Creation Time

The creation time is the time the file was created. It is based on Coordinated Universal Time.

CMS maintains the creation time only for files that reside in CMS file pools. It is stored in the file pool catalogs—not in the FSTs as basic file attributes are.

You can specify a creation time of your own choosing when you use CSL routines to create a file. Once the file is created, you cannot change the creation time.

For BFS files, this attribute has the same interpretation as for CMS record files. However, there is no equivalent POSIX attribute.

You can use CSL routines such as DMSEXIFI (Exist - File) and DMSGETDX (Get Directory - Extended) to retrieve the creation time for a file. The creation time is intended for use by application programs. It is displayed by the FILELIST and LISTFILE commands through the use of the ALLDATES option.

Date and Time of Last Change

The date of last change and time of last change attributes specify when the status or attributes of an object were changed. The SFS server records these attributes for file spaces, directories, files, aliases, and external objects. The attributes cannot be controlled by a user or SFS administrator, but administrator authority is *not* required to read the attributes.

The following additional rules apply:

- The date and time of last change are updated when authorizations are granted or revoked.
- The date and time of last change for the top directory are updated when space limits are changed (storage is added or deleted).
- The date and time of last change are updated by the restore function.
- The date and time of last change are *not* updated when a file is migrated or recalled using DFSMS/VM.
- The date and time of last change for a parent directory are *not* updated when an object is added to or deleted from that directory.
- The date and time of last change for an alias are *not* updated when the base file is updated. Only Create Alias (DMSCRALI) sets the date and time of last change for an alias.
- The date and time of last change are *not* updated when commands are issued that do not alter the object. For example, granting authority to a user who already has the granted authority does not update these attributes. Nor does opening and closing a file without writing to the file. However, if existing data in a file is overwritten with exactly the same data, this is perceived by the server as an update, and the date and time of last change are updated.

For BFS files, these attributes have the same interpretation as for CMS record files. However, when stored in the catalogs these two attributes are translated into a single POSIX attribute called CTIME that consists of the total number of seconds from January 1, 1970.

You can use CSL routines such as DMSEXIST (Exist), DMSEXIDI (Exist - Directory), and DMSEXIFI (Exist - File) to retrieve the date of last change and time of last change for a file. If you open a directory with the FILEEXT intent on the DMSOPDIR (Open Directory) routine, you can use routines such as DMSGETDI (Get Directory) and DMSGETDX (Get Directory - File Extended) to retrieve these attributes. They are also displayed by the FILELIST and LISTFILE commands through the use of the ALLDATES option.

How CMS Manages Files

CMS manages all files by continually updating various control blocks. Each accessed file mode is represented in storage by a control block called the Active Disk Table. This control block describes an accessed minidisk or SFS directory. Each open file is represented in storage by a control block called the Active File Table. This table contains the read and write pointers to a file, plus other status information related to the file. The File System Control Block (FSCB) is a user parameter list that holds information about an operation on a file. Here are some examples:

- The record format, whether it is F-format or V-format, is information maintained in the FSCBRECF field. The default format, if you do not specify it for an operation, is F-format.
- A record number is maintained in the FSCBAITN field. The record number is the relative number of the next record to be read or written. The default value for this field is 0 which, when you read a file, indicates that records are read sequentially. When you write information to an existing file, a 0 indicates that records are written sequentially following the last record in the file. When you write information to a new file, a 0 indicates that records are to be written at record 1.
- A buffer address and size is maintained in fields FSCBBUFF and FSCBSIZE, respectively. When reading or writing records, you need to supply a buffer address where a record is to be read or written. The buffer needs to be large enough to accommodate the longest record you expect to read or write.
- The number of records to be read or written is maintained in the FSCBANIT field. The default number of reads or writes for an operation is 1. This number can only be 1 for V-format files.
- The actual number of bytes read is saved in the FSCBNORD field. If you are reading a variable-length file, you can use this information to determine the size of the record read.
- A read pointer (FSCBRPTR) and a write pointer (FSCBWPTR) are saved after you open a file.

Manipulating CMS Files

In general, if you would like your application to manipulate CMS files, your application will follow these steps:

1. Open a file, specifying what you intend to do with the file. When you open a file that does not exist, you will be creating a new one. At this step you need to define the attributes of the file, such as format, name, and type, and for SFS files, recoverability and overwrite (shadowing) also.
2. If the file did not exist, you can write one or more successive records to an open file from a user specified buffer. If the file does exist already, you can read records from it and place the records into a user specified buffer OR you can write records to the file from a buffer or both.
3. Close the file. Closing the file makes the file available for other processing and frees up any resources you were using for that file.
4. Commit, or save on disk, all updates that were made.

Empty Files

Empty files, that is, files with no records, can be created in an SFS directory. (The directory must be in a file pool managed by an SFS server at the z/VM Release 1.1 level or higher.) They can be created using any of the following:

- CREATE FILE command
- ERASE command with the DATAONLY option
- FFILE and SSAVE subcommands of XEDIT
- DMSCRFIL CSL routine
- DMSOPEN CSL routine with the ALLOWEMPTY parameter
- DMSERASE CSL routine with the DATAONLY parameter
- DMSOPDBK CSL routine with the ALLOWEMPTY parameter.

Such empty files retain all aliases, SFS authorizations, and other file attributes they have been assigned.

Logically Sparse Files

A logically sparse file is a file that contains sparse records. For files with fixed-length records, you can write a record with a position number more than one greater than the number of the last record. For example, if the last record in the file, DOG DATA, has a position number of 55, you can write a record with a position number of 60. Records 56, 57, 58, and 59 are called sparse records.

Note: Only files with fixed-length records can contain sparse records.

Sparse records are not written to a file. However, if you open a file that contains sparse records, you can write to a record that was previously sparse. For example, you could write a record to position number 57 in the file, DOG DATA.

If you try to read a sparse record, it will be retrieved as all X'00' bytes. In fact, reading a sparse record will appear the same as reading a record that was actually written with X'00'.

Structurally Sparse Files

Structurally sparse files are files that contain sparse blocks. Sparse blocks are not physically stored in a file. They are data blocks or pointer blocks that contain all binary zeros.

Note: You should not write programs that depend on sparse blocks. The way that the CMS file system handles sparse blocks may change.

Sparse blocks are created two ways:

- If records are never written to a certain block in a file, then the block will be sparse. This applies to only F-format files.
- If a block contains all binary zeros, then it will be sparse. This applies to both F-format and V-format files.

Replacing Records

In any file, a record may be replaced, by writing over it, so long as its length is not modified. You cannot replace a F-format record with a different length. However, the system will let you replace an EDF V-format record with a record of a different length. You can replace a SFS V-format record with a record of a different length, only when you use the File System macro interface to replace the record. The CSL routines do not allow it.

Note: The consequence of replacing with a different length is that the rewritten record becomes the last record of the file: all records after the rewritten record would be deleted from the file.

When a CMS disk is accessed, a copy of the file directory is brought into the user's virtual storage. Whenever an output file is closed, changes to the file directory are made to the copy in storage. When the last opened output file is closed on a minidisk or the last opened file is closed on a work unit, the changes are written to disk.

EDF Data Integrity

When an existing EDF file is updated and the file is not update-in-place, a shadowing method of updating takes place. Only changed CMS blocks actually get written to disk. Each changed data and pointer block gets assigned to a new block on disk. Contents of the blocks are thus shadowed somewhere else in storage. This assists in maintaining some of a file's integrity.

When modifying an existing update-in-place minidisk file (file mode number 6), existing records are written back to their original location on disk. The updates are not shadowed as with other files. This saves DASD space that would otherwise be needed temporarily for shadowing, but may compromise data integrity.

When updating an existing file mode number 6 file, it is possible to damage the file, or even the entire minidisk on which it resides. This damage occurs when some of the updates made to the file or disk by an application are updated in place, but CMS terminates (as defined below) before it can write all of the data to disk.

File damage is possible when some of the data which was written by an application to an existing file mode number 6 file was not updated on disk because of buffering done within the CMS file system. It is possible that not all of the records the application wrote will be updated on disk. It is also possible that some records may be partially updated on disk.

Corruption of the minidisk is possible when an existing file mode number 6 file is extended (as defined below) and CMS is terminated before all the disk information kept in storage is written to disk. In this case, inconsistencies can arise between the file structure and the disk structure which would allow the file to use disk blocks which are not allocated to it. This can lead to additional disk damage, where two files are temporarily using the same disk block. This type of damage may go undetected for long periods of time and may surface long after the original damage has taken place. (In fact, the original file mode number 6 file, which was involved in the damage, may no longer exist.)

Damage is made more likely by applications which leave the updated file mode number 6 file open for long periods of time, or which delay the commit of the disk by keeping other output files opened on the same disk. Note that for any update-in-place processing to occur, output must be written directly to the file as a file mode number 6 file. Many commands, such as COPYFILE with the REPLACE option and XEDIT, do not perform update-in-place processing because they use a separate work file. However, COPYFILE with the APPEND option and EXECIO do perform update-in-place processing when the target file is a file mode number 6 file.

Note that for a file mode number 6 file created by the CMS RESERVE command, minidisk damage as described above cannot occur because the file cannot be extended (it fills the entire disk). Also, because the CMS RESERVE command creates the file with a record length equal to the disk block size, individual records cannot be partially updated. However, some records which were successfully written by an application may not be updated on disk due to buffering, as mentioned above.

The definitions referred to above are:

CMS Termination

This means any abrupt end of CMS execution which does not allow the CMS file system to complete processing. This includes:

- A malfunction of the entire system
- A malfunction of CMS in the user's virtual machine which requires CMS to be re-IPLed
- A re-IPL by the user by invoking the CP IPL command while the application is running
- A re-IPL of CMS from within an exec or application which invokes the CP IPL command
- A forced logoff of the user's virtual machine
- A logoff by the user by invoking the CP LOGOFF command while the application is running
- A logoff from within an exec or application which invokes the CP LOGOFF command.

Note that this does not include cases where CMS abend recovery occurs, ending in a READY message.

Extending the File

This means any operation which causes new disk blocks to be added to an existing update-in-place file, even though the current processing is being treated essentially as update-in-place. This includes:

- Appending records to the end of an existing file. Note that some append operations may only update the last existing block of the file, but in many cases, the file will need additional blocks allocated as well.
- Changing the length of records in a V-format file. (This essentially causes truncation of the file to the record preceding the one being written and then an append to the end of the file.)
- Rewriting existing records to the file which cause previously sparse blocks (all binary zeros) to be allocated. Sparse blocks may exist either because the record in question was never previously written to the file or because the records written caused one or more entire disk blocks to contain binary zeros.

SFS Data Integrity

SFS also maintains new copies of changed blocks. When an existing SFS file is updated and the file's overwrite attribute is not inplace, each changed data and pointer block gets assigned to a new block on disk. When you specify an SFS file's overwrite attribute as inplace, changes to the file will be written back to the original location where possible.

In addition, SFS provides a *rollback* capability that permits a recoverable file to be restored to a previous state, even though updates to the file have occurred. If you specify the NORECOVER file attribute, updates to the file will not be rolled back, and will be committed if possible.

For more information on the overwrite and recoverability attributes, see the *z/VM: CMS Application Development Guide*.

Manipulating BFS Files in CMS

To manipulate BFS files, your application will follow these steps:

1. Open a file, specifying what you intend to do with the file. You cannot create new BFS files using this interface, so you cannot open a file that does not exist.
2. Read records from the file and place them into a user-specified buffer, or write records to the file from a buffer, or both.
3. Close the file. Closing the file commits the changes, makes the file available for other processing, and frees up any resources you were using for that file.

For more information about manipulating BFS files in CMS, see the *z/VM: CMS Application Development Guide*.

Empty or Sparse BFS Files

Empty or sparse BFS files may exist. They have the same characteristics as empty or sparse CMS record files.

Using Multiple Work Units

Using multiple work units for updates to BFS files and directories uses the least amount of CMS file pool server machine resources and helps keep files and directories available for other users. In addition, multiple work units are required for doing work on both SFS files and BFS files in the same file pool. Updates to SFS files and BFS files in the same file pool cannot be included on a single work unit.

BFS Data Integrity

Because BFS files are stored in CMS file pools, the file pool server maintains new copies of changed blocks for BFS files. All BFS files have an overwrite attribute of NOTINPLACE. When an existing BFS file is updated, each changed data block gets assigned to a new block on disk. Therefore, a user sees a consistent version of the file from open to close.

In addition, all BFS files have a recoverability attribute of RECOVER. Changes to an open BFS file can be rolled back to the previous state. However, when the file is closed the changes are committed. BFS files do not participate in Coordinated Resource Recovery (CRR).

Application Interfaces

There are three interfaces (sets of routines or macros) that allow applications to create and modify CMS files and BFS files:

- Record file system CSL routines
- FS macros
- OpenExtensions CSL routines

Additionally, CMS simulates OS and DOS/VSE macros that can be used to manipulate CMS files. See Part 5, “ OS/MVS Simulation,” on page 315, for more information.

Record File System CSL Routines

A program interface which is available for both minidisk and SFS files is a call interface composed of callable services library routines in VMLIB CSLLIB. The routines can be called from high-level languages as well as assembler. On the routines, you can specify file names, directory names, file mode letters, name definitions, length of data for operations, buffer addresses and buffer sizes for I/O, and position in a file to perform I/O. These routines do not automatically commit changes when you close the SFS file. All changes to SFS files and directories must be explicitly committed or rolled back. Conversely, changes to minidisk files are made automatically for both the CSL and FS routines when the last file opened for output is closed. The CSL routines provide recovery if the system or program abends.

The record file system CSL routines also work, with limitations, on BFS files. This support is primarily for administration and system-managed storage. When using these routines to manipulate BFS files, you can specify file names, directory names, and name definitions, but not file mode letters. BFS files and directories are not accessed.

Note: In the context of manipulating BFS objects in CMS, the term "file" refers only to a BFS regular file. Other types of BFS files cannot be manipulated by CMS record file system functions. The term "directory" refers only to the top directory in a BFS file space (that is, the byte file system itself). BFS subdirectories are not equivalent to SFS subdirectories and cannot be manipulated by CMS record file system functions.

Changes to a BFS file are committed when the file is closed. The file cannot be closed without committing the changes, and the changes cannot be committed before closing the file. However, changes can be rolled back before the file is closed.

The record file system CSL interface and examples of its use are described in the *z/VM: CMS Application Development Guide*.

FS Macros

FS Macros are available that allow an assembler program to manipulate minidisk and SFS files. FS macros cannot be used to manipulate BFS files. On the FS Macros, you can specify attributes such as record format, buffer address and buffer size for performing I/O, the record number to use for the next I/O operation, and the number of records you will read in the next read operation. The FS macros also accept a *form* parameter, which specifies whether an extended format file system control block is to be used[6]. You should always use extended format for the greatest efficiency. An extended format FSCB lets you specify a value up to (2^{31}-1) for record number and number of records to be read. If you do not specify extended format, then the record number values cannot exceed 65535. The FS Macros and examples of their use are discussed later in this chapter.

OpenExtensions CSL Routines

The primary programming interface for manipulating BFS files and directories is a call interface composed of a set of CSL routines in the VMMTLIB callable services library. VMMTLIB is included in the CMS nucleus. The OpenExtensions CSL routines are intended for use by language runtime environments. They can also be called from assembler and REXX programs. For more information, see the *z/VM: OpenExtensions Callable Services Reference*.

Open Intent

The record file system CSL interface and the FS macro interface let you specify the intent of operations when you explicitly open the file. On FS Macros (FSCB and FSOPEN) you use the OPENTYP= parameter. On the CSL routines (DMSOPEN) you just specify the intent. Valid intents for SFS files and minidisk files are NEW, READ, REPLACE and WRITE. Valid intents for BFS files are READ and REPLACE. NEW opens a file for output (invalid if a file already exists). READ opens a file for input. REPLACE opens a file for output (valid if a file already exists) and indicates the intent to entirely replace the existing file. WRITE opens a file for update. If the SFS or minidisk file does not exist, REPLACE and WRITE are treated just like NEW. For a BFS file, REPLACE is valid only if the file already exists. You cannot create a BFS file using the CMS record file system interface. Regardless of the open intent, a read operation is always valid after the file has been opened. However, records that existed before the file was opened cannot be retrieved if the open intent is REPLACE.

Extended File Attributes

You can also specify the recoverability, overwrite, creation date, and creation time attributes of an SFS file with DMSOPEN and DMSOPDBK. The creation date and creation time attributes can be specified only if the file is new. The recoverability attribute specifies whether you want uncommitted changes rolled back in the event of an application initiated rollback (RECOVER) or you want them committed to DASD where possible (NORECOVER). The overwrite attribute specifies whether file writes are to be shadowed, so that the user sees a consistent version of the file from open to close (NOTINPLACE) or the file writes are to be made in place (INPLACE). Also, you can override the recoverability and overwrite attributes of a file by specifying the OPENRECOVER parameter on DMSOPEN.

For BFS files, the only recoverability and overwrite attributes allowed (on an intent of REPLACE) are RECOVER and NOTINPLACE. The create date and create time attributes do not apply to BFS files because BFS files cannot be created using this interface.

You cannot specify a creation date and time for files created by using FS macros. The system determines the creation date and time for you. You cannot assign recoverability or overwrite attributes to a specific file using FS macros, but if you need to, you can control the default attributes for files of a specific

[6] When EDF files were introduced, the parameter list used by file system control blocks was enhanced. To make use of this extended format, you need to specify FORM=E on the FS macros.

file mode number using the DMSPUSHA CSL routine. For details, see the *z/VM: CMS Callable Services Reference*.

Caching

The CMS record file CSL interface and the FS macro interface have a CACHE option for explicit opens of files. Use the CACHE option to indicate whether caching of multiple data blocks is to be performed for a file.

The cache size may be different for SFS and minidisk files. These values may be set at system initialization time. (The SFS default cache size is 20KB, the minidisk default cache size is 8KB.) Note that the SFS cache size also applies to BFS files, which are stored in SFS.

Caching could reduce the number of actual I/O operations performed on a file. If improving I/O performance is not a concern, you should specify CACHE=DEFAULT (on the FS Macros) when explicitly opening a file. Specifying CACHE=DEFAULT will allow the file system to determine whether to cache multiple data blocks, based on the file's characteristics and the actual or anticipated accesses to the file. If you do not specify CACHE, then the system will choose whether caching is appropriate. Not using the CACHE parameter on a CSL routine is equivalent to CACHE=DEFAULT on an FS Macro. Generally, you would want to use caching when the probability is high that I/O operations will be sequential. More specifically, use caching when:

- Reading many small (less than cache size) records one at a time.
- Writing many records one at a time sequentially forward when the total amount of data to be written is significantly larger than the disk block size.
- Reading or writing is not strictly sequential, but most reads and writes of records are clustered.
- Directly accessing a small (less than cache size) file.
- Accessing records that are close together (no more than the size of the cache apart).

Do not use caching when:

- User’s virtual storage is severely constrained.
- The entire file is being read with a single FSREAD into the user’s buffer.
- Accesses to the records in a large file are neither sequential nor clustered (reads or writes to the file are usually more than the size of the cache apart).
- The amount of data being transferred on each read or write is more than the cache size.
- The file is being accessed sequentially backward.

Compatibility Issues Between Interfaces

There are several considerations you must make if you use FS macros on files stored in the Shared File System. This will depend on whether your file is shared or not, whether your program tries to replace an SFS file, among other things. You may also be using programs that use both FS macros and CSL routines to access minidisk and SFS file data.

For more information on manipulating files in the Shared File System, see the *z/VM: CMS Application Development Guide*.

Using Programs on Non-Shared SFS Files

Programs written for minidisk files can manipulate non-shared Shared File System files without modification. Your programs will work the same as they did before if:

- The files they use are not concurrently updated by other users.
- There are no aliases associated with the files.
- The files and directories are not locked by another user. They are also not locked by your user ID in share mode.
- There is no dependence on minidisk addresses, such as 191.

- There is no concurrent write sharing to the file space (that is, file pool and user ID) among users.
- All SFS files being updated by your application reside in CMS file pools maintained by file pool servers that are at Release 2.1 or above. (Note that the files can reside in more than one file space, and in more than one file pool.)
- They check for non-zero return codes after each file system call (for example, FSOPEN, FSWRITE, FSCLOSE).
- There is no dependency on internal control blocks or internal routines.
- There are no empty files in your directory.
- There are no external objects in your directory.
- They create and modify SFS files exclusively using:
  - FS macros,
  - EXECIO, or
  - OS Simulation WRITE/PUT.
- They do not explicitly acquire work unit IDs for SFS processing. In other words, your application does not use the DMSGETWU, DMSPUSWU, or DMSPOPWU CSL routines to manipulate work unit IDs.
- SET RECALL is set ON to insure that implicit file recall will be done if any of the files referenced have been migrated by DFSMS/VM.

However, because the programs are unaware of directory names, your program can reference files only by file name, file type, and directories accessed as a file mode. You must issue an ACCESS command for the directory in which the file resides. After accessing the directory, you can run the program without modification.

Note: Even when you have not granted authority to another user (you don't want to share), any user with administrator authority can change any of the files, directories, or your space allocation, which could cause an application to fail.

Programs that call low level file system routines or reference file system control blocks may require code changes to work with CMS files in the Shared File System. In certain cases, programs that reference system control blocks may require reassembly, code changes, or both to work, even for CMS minidisk files.

Using Programs Written for Minidisks on Shared SFS Files

Programs that use FS macros, OS simulation, DOS simulation, and the EXECIO command will also work correctly in certain file sharing situations.

SFS directory control directories have sharing characteristics similar to those of minidisks. Programs are likely to need few modifications (if any) to work correctly on files residing in directory control directories.

SFS file control directories permit more concurrent sharing of files than minidisks. Programs written for minidisks will work correctly only in certain file sharing situations, such as:

- Reading shared files which are not locked or being updated by other users.
- Updating (but not replacing) shared files that meet ALL of the following criteria:
  - The file is not concurrently locked or open for update by other users.
  - The file is being referenced through a directory accessed read/write.
  - The user is authorized to write to the file.

  (For more information about programs that update and replace files, see "Modifying Programs That Replace Files" on page 123.)

Regardless of whether directory control or file control directories are being used, programs written for minidisks may act differently in certain file sharing cases, and might require changes to work correctly. For example, you might need to change your program for:

- New return codes

- File space usage considerations
- Files shared through aliases
- Other user's directories accessed as read/write
- Reading shared files that are locked by other users
- Replacing shared files
- Update-in-place processing.

The conditions described apply to programs coded in high-level languages as well as programs coded in assembler language.

New Return Codes

The program could run into an error condition unique to SFS (such as the file pool not being available). Several new messages and return codes are provided to reflect these errors. You can find a list of the SFS error messages and file error messages in the "System Messages" chapter of the *z/VM: CMS Commands and Utilities Reference*. For a list of return codes that you may encounter using SFS, see *z/VM: CMS Callable Services Reference*.

The program may need to check error conditions which did not previously occur. Programs that handle any nonzero return code should not be impacted.

File Space Usage Considerations:

A CMS program might assume that there is sufficient space available to make all file updates permanent. When writing to SFS files in CMS file pool servers running at Release 2.1 level or higher using FSWRITE, EXECIO, or OS Simulation WRITE/PUT, write operations will fail when a file space full condition is detected. [7] However there is no guarantee that there will be enough space to commit the changes when the file space is being shared. Your program may detect that there is insufficient space to commit changes when the last output file is closed. At this point, the system may have rolled back all file updates.

If your program is coded to monitor the SFS file space threshold exceeded indicator, it is more likely to be able to anticipate a file space full condition in a file sharing environment. (See "Monitoring SFS Filespace Threshold" on page 138.) However, this is only available when using the FS macro interface.

Files Shared through Aliases

A CMS program might assume that if it can write to a file in a file mode, it can write to any other file in that file mode. In SFS, that file mode actually represents some directory. If the directory is a file control directory, it might contain aliases to base files on which you have read-only authorization. A program that assumes it can write to the alias will receive an error and may fail.

Aliases cannot be created in directory control directories, so there is no problem with aliases referring to read-only base files. Furthermore, to access a directory control directory in read/write status, you must have DIRWRITE authority. DIRWRITE authority lets you write to any file in the directory. So, a program can safely assume that if it can write to a file in a directory control directory, it can write to any other file in that directory.

Because users can grant individual file authorizations on files in file control directories, programs operating on those files may require modifications to handle authorization errors. These authorization errors are typically reflected to your program as return code 28 (which is also returned when a file is *not found*).

Other User's Directories Accessed as Read/Write

You can access another user's directory in read/write mode by using the FORCERW option on the ACCESS command. For directory control directories, DIRWRITE authority is needed for the access to succeed. So,

[7] FSWRITE and EXECIO will not fail for a file space full condition when writing to pre-Release 2.1 file pool servers. For pre-Release 2.1 servers, OS Simulation WRITE/PUT operations will fail when the file space threshold is exceeded.

if you access another user's directory in read/write mode, you can be sure that you can write to all the files in the directory, as well as create new ones. This is compatible with the way minidisks work.

For file control directories, however, only READ authority to the directory is needed for the ACCESS to succeed. WRITE authority is not required. While this lets you issue commands that require a read/write file mode, it also might cause compatibility problems with your programs. Your program might, for example, assume that is can create new files on a file mode if it can write to any file in that file mode. Because only READ authority to the directory is required for a read/write ACCESS, you program may fail when it tries to create a file.

Because the owner of the file control directory may write to files for which you have WRITE authority, your program may also encounter sharing conflicts. For information on the ACCESS command, see the *z/VM: CMS Commands and Utilities Reference*.

Reading Shared Files That Are Locked

When a program is reading a shared file control directory or a shared file that resides in a file control directory, another user could have a lock on the object, preventing the program from reading an object that it knows exists. In this case, message DMS1137E or DMS1138E and return code 31 or 70 are presented. The program may require modification if it does not handle all nonzero return codes.

This problem does not occur with directory control directories or with the files that reside in directory control directories. Only UPDATE locks are permitted on directory control directories and files within directory control directories. After an UPDATE lock is created on a file, only the user who holds the lock can access the directory in read/write mode and change the file. This prevents problems with programs designed to work on minidisk files.

When a directory control directory is locked, not even the person who holds the lock can access the directory read/write.

Replacing Shared Files

When modifying a minidisk file in its entirety, the file is typically erased first and then rewritten sequentially. Two typical approaches used are:

- When recoverability is desired, the output is placed in a temporary file, the original file is erased, and the temporary file is renamed to the original file.
- When recoverability is not needed, the output file is simply erased, and then rewritten from beginning to end.

In either case, erasing the file (using the FSERASE macro or ERASE command) destroys the sharing of the file:

- If the file was being accessed through an alias, the alias is erased and a new file with the same name as the base file is created. Note that the original base file is still intact, but bears no connection to the new output file.
- If the file being replaced was simply a base file, erasing it will result in the loss of all aliases to the file. If the base file resides in a file control directory, any individual authorizations (other than NEWREAD and NEWWRITE) granted on that file would also be lost. Even though the file has the same name as the file it replaced, it is essentially a new file. You must reestablish all the dropped aliases and, for files in file control directories, grant the authorities lost when the file was erased.

This problem pertains mainly to the DOS simulated access methods, the FS macro interface (FSERASE), or programs and execs which use the ERASE command. High level language applications which use OS simulated access methods to open output files would not encounter this problem, because the CMS simulation of the OPEN macro will not result in an erase of the file when it is an SFS file. The COPYFILE command with the REPLACE option also retains aliases and authorities.

For the reasons described above, write sharing may not work with existing non CSL programs that operate on SFS files. Changes may be required either within the programs, or in the manner in which they are used to properly preserve sharing of output files.

Modifying Programs That Replace Files

Programs that replace files by erasing and rewriting them may be modified in any of several ways to avoid the previously-mentioned problems:

- For the case where a temporary file is used, change the program to issue a COPYFILE command or the DMSFILEC routine (both with the REPLACE option) to copy the temporary file to the original file, and then erase the temporary file. The COPYFILE command and DMSFILEC routine preserve aliases and authorities of the output file.
- For the simpler erase/rewrite scenario, change the erase to an FSOPEN with OPENTYP=REPLACE specified. This will work for both minidisk files and SFS files. Alternatively, you may change the program to use a temporary file and handle it as above, using COPYFILE with REPLACE.
- Use the Shared File System routines for SFS output files, and use the macro interfaces for minidisk output files.
- Do not use the program directly on shared output files. Copy the output files to temporary, private files. Run the program against the private files. Finally, copy the output files back to the actual files, using COPYFILE with REPLACE.

Update-in-Place Processing

If your application uses update-in-place files, and you want to use that application in SFS, you can make SFS files update-in-place:

- If the output file(s) already exist, you can change the overwrite attribute of the file(s) with the DMSCATTR (Change Attributes) CSL routine or with the FILEATTR (File Attributes) command.
- If your application creates new files, you can use the DMSPUSHA CSL routine to specify update-in-place for files with a certain file mode number.

These commands and CSL routines are described in the *z/VM: CMS Commands and Utilities Reference* and *z/VM: CMS Callable Services Reference*.

Using Non-CSL Statements or Macros with SFS Files

There are four ways to access SFS files using non-CSL statements or macros:

1. Using high-level language statements for reading and writing files, such as READ/WRITE in FORTRAN, or GET/PUT in PL/I
2. Using the CMS FS macros, such as FSOPEN, FSREAD, and FSWRITE
3. Using OS and DOS macros that CMS simulates
4. Using the EXECIO command.

For more information on the EXECIO command, see the *z/VM: CMS Commands and Utilities Reference*.

Using High-Level Languages with SFS Files

To operate on files stored in file pools using a high-level language such as FORTRAN, COBOL, or PL/I, you do not need to do anything special. Code your program as you would code it to access a CMS file stored on a minidisk.

SFS will automatically use the current default work unit ID when processing the file I/O commands. At the end of the CMS command (when you see the `Ready;` message), SFS will automatically issue a COMMIT for the program. (For more information on using work units in application programs, see the *z/VM: CMS Application Development Guide*.)

Using FS Macros with SFS Files

Programs coded with FS macros (File System macros) work on non-shared files stored in file pools without modification. If you want to use these programs on shared files, it may be possible to do so with minor changes or no changes, while still using the macro interface. See the discussion in "Using Programs Written for Minidisks on Shared SFS Files" on page 120 for details.

SFS will automatically use the current default work unit ID when processing the FS macros that operate on files stored in file pools. Commit of data occurs either at the end of the CMS command (Ready; message), or on an FSCLOSE (or FINIS command), provided no other files or directories are opened at the time the FSCLOSE is issued.

FSOPEN

FSOPEN supports an OPENTYP option which describes the type of input/output activity to be performed: READ, WRITE, NEW, REPLACE. The OPENTYP option applies to files on minidisks and SFS files.

For SFS files, the OPENTYP option is useful as follows:

- It assures that the file is not already open with the file system macro interface.
- It allows the user to specify the type of operation to be performed on the file. This allows the file system to properly open and lock the file to safely allow multiple concurrent access to the file, and provide the user with a consistent image of the file from open to close. Without the OPENTYP parameter, the FSOPEN macro only performs an existence check (equivalent to FSSTATE), and does not actually open and lock the file, which could result in a change to the file by another user between the FSOPEN and the first FSREAD, FSWRITE, or FSPOINT to the file.
- For output files, the OPENTYP=REPLACE option lets the file be replaced without loss of authorities and aliases. Prior to VM/SP Release 6, a file would typically be erased and rewritten to replace it. For more information on replacing shared files, see "Replacing Shared Files" on page 122. In the Shared File System, erasing a file causes all authorities and aliases to the file to be lost. (Note that authorities are lost only on files in file control directories—authorizations cannot be granted on individual files within directory control directories.)
- For OPENTYP=READ, it prevents subsequent writes to the file.
- For OPENTYP=NEW, WRITE, or REPLACE, it assures that the file is on a directory accessed R/W.

For minidisk files, the OPENTYP option is useful as follows:

- It assures that the file is not already open.
- For OPENTYP=REPLACE, it erases the file at open time, making it unnecessary for the application program to do so.
- For OPENTYP=READ, it prevents subsequent writes to the file.
- For OPENTYP=NEW, WRITE, or REPLACE, it assures that the file is on a minidisk accessed R/W.

FSWRITE

An FSWRITE to an alias for which the user is only authorized to read will result in an error.

If SFS files reside in a CMS file pool server at a release level earlier than Release 2.1, FSWRITE allows you to temporarily write more blocks than are allocated to the file space. File space limits are enforced when the last active output file is closed. See "Monitoring SFS Filespace Threshold" on page 138 for a discussion of how you can monitor file space usage using FSWRITE.

In a CMS file pool server that is at a release level of Release 2.1 or later, FSWRITE will fail when it detects that the file space limit is exceeded.

FSWRITE will truncate a variable-length file in the Shared File System when an existing record has its length modified. No error codes are returned. If you are using the Shared File System and do not need to use minidisk files, you should consider using the SFS DMSWRITE routine.

FSCLOSE

You must check the return code in order to verify the disposition of file updates. A non-zero return code from FSCLOSE may mean that the work unit has been rolled back and file updates have been discarded.

Using OS and DOS Macros with SFS Files

Internally, OS and DOS macro simulations use the same CMS I/O routines that FS macros do, so the OS and DOS macros will work on SFS files. The same discussions about compatibility apply, as described in "Compatibility Issues Between Interfaces" on page 119.

Note that the problem with lost aliases and authorities common to programs which use the FS macro interface to replace files does not apply to OS access methods, but it does apply to DOS. OS simulated access methods do not erase SFS output files, rather, they open them with replace intent.

SFS will automatically use the current default work unit ID when processing OS or DOS macros that operate on files stored in file pools. At the end of the CMS command, SFS will automatically issue a COMMIT for the program.

OS Simulation Usage Notes

- As long as no read/write sharing is taking place (that is, one user reads a file while another user writes to it), the version of all data sets is guaranteed to remain constant to the OS application. Only those changes made by the application itself will be seen. This guarantee does NOT exist for a read/write sharing environment unless you first create a lock on the directories that have files affected.
- All SFS directories that are used for output by an OS application must be accessed as read/write.
- A program that uses OS simulation may experience unwanted results such as an unexpected view of the file or a file sharing conflict under the following circumstances:
  - The default SFS work unit is changed during the execution of the program, or
  - A file used by an OS simulation application is already open at entry to that application and the default work unit on which the file was opened is different from that used during the application, and the application closes and then reopens the file. The open prior to entry was performed through FSOPEN, EXECIO, or a previous OS OPEN.
- If a data set whose FILEDEF specifies a file mode of '*' is opened for output, all read/write file modes will be searched for the file. If found on any of those file modes, that file will be used. Otherwise, the file mode will default to A1. This can cause performance problems if large read-write disks or directories must be searched.
- Each time a CLOSE is issued for a DASD DCB without the TYPE=T parameter, the file represented by that DCB is closed with the FINIS command.
- When you execute programs with OS macros that write to SFS files, you should ensure that there is an ample supply of unused blocks in the file space to which you are writing. Otherwise, your program could abnormally stop when your program tries to close the file. You might also consider limiting write access to the file space so that other users do not consume the blocks you require for your application.

Using OS/MVS interfaces to write to an SFS file will generally result in a disk full error when the SFS file space limit is exceeded. However, the disk full condition will be detected when the file space threshold is exceeded for an SFS server running at release levels prior to Release 2.1.

Using Record File CSL Routines with BFS Files

The CMS record file system CSL routines provide limited support for BFS files, primarily for administration purposes. For example, the record file CSL interface works only on existing BFS files. For more information on manipulating BFS files in CMS, see the *z/VM: CMS Application Development Guide*. Full support for BFS files is provided by the OpenExtensions CSL interface. See the *z/VM: OpenExtensions Callable Services Reference*.

Committing Changes

When operating on files in SFS, all commits are made based on the work unit. Changes can be committed when the file is open on the same work unit or when the file is closed (using FSCLOSE). Using FSCLOSE on the last file that was opened within a work unit (or the last output file opened), performs a commit.

Committing your changes while the file is open (using the DMSCOMM routine), enables the writer to see the new level of changes without closing and re-opening the file. SFS files can be rolled back using the DMSROLLB routine. This rolls back changes to all recoverable files in the work unit.

Rollback is not supported for minidisks. Minidisk file updates become permanent when the last open output file on the file mode is closed.

Changes to a BFS file become permanent when the file is closed. You cannot commit changes before the file is closed, and you cannot close the file without committing the changes. Before the file is closed, changes can be rolled back using DMSROLLB.

Mixing CSL and Non-CSL Statements

Within a single program you can mix both non-CSL statements (or macros) and CSL routines (callable services library routines). There are several reasons you might want to do this. The most common are:

1. You have a program written prior to VM/SP Release 6 that is being used to operate only on files in file pools, and you want to enhance subroutines within the program to take advantage of SFS routines.
2. You want to code a program that has two file I/O routines: one for files stored on minidisks, and another for SFS files stored in CMS file pools. Many of the CSL routines operate on minidisk files as well as SFS files. An application may use only CSL routines to operate on both types of files. There will be occasions though where your application can issue a function only for a minidisk (FORMAT) or to SFS (CREATE ALIAS). The program would use the DMSQFMOD routine (Query Filemode) or DMSVALDT (Validate File Name) to determine whether the file was on a minidisk or in a file pool.
3. You want to create or update a file in another user's directory and you want to use CSL routines for direct file reference rather than access the directory as a file mode.

Once a file is opened using one method, you must continue to use that method to read, write, or close that image of the file. After the file is closed, you can re-open it using a different method.

You cannot, for example, open a file using FSOPEN, read it using a CSL DMSREAD routine, and then close it using FSCLOSE. Similarly, you cannot use CSL DMSOPEN and DMSCLOSE routines, and an FSWRITE macro to write to this image of the file. Mixing methods at that level will cause an execution error in your program.

You can operate on files simultaneously using both methods. The following describes how each method can be used for both minidisk and SFS files.

- When using DMSOPEN:
  - For an SFS file, you can open the file more than once for input (read) AND only once for output (new, write, or replace).
  - For a minidisk file, you can open the file more than once for input (read) OR once for output (new, write, or replace).
- When using FSOPEN:
  - For both a minidisk and SFS file, the file may only be opened once for either input or output.

Notice that CSL and non-CSL statements use the same default work unit ID. If you want the CSL routines to execute in a different work unit than the non-CSL statements, you will have to use the DMSGETWU routine (Get Workunitid).

It is possible to open an SFS file that is already open. It does not matter whether the file is opened using CSL or non-CSL statements. That is, you can open a file using the FS macros and then open the same file using CSL routines.

Note that CSL and non-CSL statements handle file space limits differently. You may write past your file space limits temporarily using CSL routines, but not via FS macros, EXECIO, [8] or OS Simulation

[8] You may write past your file space limit temporarily using FSWRITE and EXECIO for SFS files in a in a pre-Release 2.1 CMS file pool server.

WRITE/PUT macro operations. If you are writing past your file space limit and mixing interfaces, your updates may be rolled back.

Which Method Should You Use?

When coding a new program, you will need to decide whether to use CSL routines or not. If you choose not to use CSL routines, you will have a choice of high-level language statements, FS macros, OS macros, or DOS/VSE macros that CMS simulates.

Some important points to remember are:

- OS macros, DOS/VSE macros, and FS macros are assembler language macros.

  To use assembler language macros directly you must code in assembler language. If you code in a high-level language such as FORTRAN or PL/I, the interface is generated automatically but with less control than you would have using the assembler language macros directly.

  CSL routines, on the other hand, can be called from any high-level language as well as assembler programs and REXX execs.
- OS macros, FS macros, and EXECIO may only be used to write to files in a directory accessed as read/write.

  To write to a shared file using these interfaces, one of the following must be true:

  1. The file is in your directory, and it is either a base file or an alias of a base file to which you are authorized to write. DOS macros may only be used to write to files you own; you cannot write to an alias.
  2. The file is in another user's directory and you have used the FORCERW option of the ACCESS command to access the directory.

  For more information on accessing directories, see *z/VM: CMS Commands and Utilities Reference*.
- Many CSL routines work on both minidisk and SFS files.

  Your program may be coded using CSL routines so that it can operate on both minidisk and SFS files. Using SFS files gives applications a greater range of functions that can be used (workunits, controlled commits, direct file access) over minidisks. However, many desirable elements from the CSL routines (use by high level languages, namedefs) may also be used on minidisk files as well.
- Many CSL routines work, to a limited extent, on BFS files. This support is primarily for administrative purposes. It allows programs that provide system services, such as backup, to operate on all files. FS macros do not provide any BFS support.
- CSL routines allow you to do more with files than you can do with non-CSL statements or macros. Some of those benefits are:
  - You can use a name definition (namedef) to identify a file or directory externally to your program. Using namedefs you can write a program to process different files and directories without changing the code and recompiling the program.
  - You can acquire work unit IDs and perform tasks independently of other tasks. Each work unit is independent of the other. Therefore, you can make changes to SFS files within a work unit and commit them or roll them back independently of any other work unit.
  - You have control over when work is committed when using CSL routines. You can commit changes as you complete a task. You do not have to close an SFS file or directory to save your changes.
  - You have more control over the assignment and use of file attributes specific to SFS files (like recoverability, overwrite). They can be assigned to a specific file and, in some cases, overridden while processing a specific application.
  - CSL routines provide more detailed error information. If you use CSL routines on files, you will receive return codes and reason codes that are more specific than those presented by non-CSL interfaces.
  - You do not need to access the directory before you can read or write to the file.

  - CSL routines allow you to temporarily write more file blocks than are allocated to your file space. This avoids the need for temporary space when your application uses work files that will be deleted prior to a commit of the work unit.
- SFS files eliminate the need to use temporary files when updates are being made to files. Because the SFS ensures the integrity of work units (either all changes complete successfully or none of them do), there is no need to create a temporary file and then rename it when you are sure the changes are successfully made. This enhances the application's performance.
- You can synchronize SFS file updates. By placing updates to more than one SFS file in a single work unit, you do not have to make provisions for system failures that might occur while your application is executing. If a system failure were to occur before all the changes in the work unit were made and committed, the changes are automatically rolled back the next time SFS is started. You do not need to worry that perhaps the changes to one file were made, but the corresponding changes in another file were not.
- You can read and write to SFS files that are remote from the processor that your program is running on.
- You can control which updates to keep and which to discard in the event of an abnormal termination or rollback by assigning the appropriate value to the recoverability file attribute. If an application abends, the changes to a recoverable file are rolled back and the changes to a irrecoverable file are committed.
- If an application abends, the changes to a recoverable SFS file are rolled back. You can control which updates you wish to keep and which to ignore in the event of an abnormal termination or rollback by the value of the recoverability attribute.
- SFS specific file attributes like recoverability and overwrite may be coded in CSL routines.

Using XEDIT to Access Files in Storage

XEDIT provides a set of SUBCOM interfaces to allow applications to read and write data to a file being edited. CMS uses the SUBCOM facility to allow a number of CMS commands to use an XEDIT interface to access files in storage. Applications can read or write specific records without having to go to disk or use the program stack to transfer the data to or from XEDIT. This improves performance.

CMS uses the XEDIT interface for processing the FILELIST, HELP, MACLIST, PEEK, and SENDFILE commands. The interface is invoked by specifying the XEDIT option on the LISTFILE, MACLIB, or NAMEFIND commands. This option can only be specified from the XEDIT environment.

The XEDIT interface is similar in structure to the FS macro interface of the CMS file system. The application must provide a FORM=E FSCB. The return code is returned in register 15. See the *z/VM: CMS Macros and Functions Reference* for descriptions of the FSCB and FST data areas.

Two methods of invocation are available:

CMSCALL macro
: This is the preferred method. Specify CALLTYP=SUBCOM and the address of the FSCB for the PLIST parameter.

SVC 202
: Register 1 must point to an FSCB below the 16MB line, and its high-order byte must be X'02' to indicate SUBCOM invocation.

The routines available, their entry point names, and error return codes are:

- DMSXFLST - This routine returns the characteristics of a file (RECFM, LRECL, and so forth) in an FST whose address is placed in field FSCBBUFF. It also ensures that the file is in the XEDIT ring. The return codes are:

  0
  File is in the XEDIT ring

  24
  Incomplete file ID specified

  28
  File is not in the XEDIT ring.

Note: Return codes are similar to those for ESTATE.

- DMSXFLRD - This routine transfers one record from XEDIT storage to the calling program. If RECFM=F, it may transfer more than one record. The return codes are:

  0
  Read performed

  1
  File is not in the XEDIT ring

  2
  Invalid buffer address

  5
  Number of records to read equals zero

  7
  Record format is not fixed or variable

  8
  Buffer is too small (records truncated)

  11
  Number of records to read is not equal to one for V-file

  12
  End of file.

  Note: Return codes are similar to those for FSREAD.

- DMSXFLWR - This routine transfers one record from the calling program to XEDIT storage. If the record format is fixed, it may transfer more than one record. The return codes are:

  0
  Write performed

  2
  User buffer address equals zero

  7
  Skip over unwritten records

  8
  Number of bytes is not specified

  11
  Record format is not fixed or variable

  13
  No more space is available

  14
  Number of bytes is not integrally divisible by the number of records

  15
  Record length is not the same as previous

  16
  Record format of fixed or variable is not the same as previous

  18
  Number of records to read is not equal to one for V-file

  28
  File is not in the XEDIT ring.

  Note: Return codes are similar to those for FSWRITE.

- DMSXFLPT - This routine moves the current line pointer to a record the calling program specifies. If you specify the read and write pointer as all ones (X'FFFFFFFF'), the current line pointer is returned in the FSCB. The return codes are:

0
: Point performed

1
: File not found

2
: Invalid FSCB

Note: Return codes are similar to those for FSPOINT.

When this interface is used, XEDIT determines if a file is in the XEDIT ring (active in storage) and does the processing required. The files in the XEDIT ring are always "open". You can add new files to the ring with the XEDIT subcommand and "close" files in the ring with the FILE or QUIT subcommands.

The current line pointer serves the function of both the read and write pointer of the CMS file system. If RECNO=0 is specified in a call to DMSXFLRD, the data is transferred to the calling program starting at the current line pointer. Transfer is stopped when the specified number of lines has been transferred or when end-of-file is reached. The current line pointer is advanced by one for each record transferred to the calling program. If the current line pointer was at the end-of-file when DMSXFLRD was called, no data is transferred and an end-of-file condition is returned.

If RECNO=0 is specified in a call to DMSXFLWR, new records are written starting at the line pointed to by the current line pointer. These new records replace any existing records or add new records if at the end-of-file. The current line pointer is advanced to the line following the last line written at the end of the operation. Note that writing to a record in the middle of a V-format does not result in truncation of the file from that point as it would in the CMS file system. Truncation (or spilling when SET SPILL ON|WORD) may occur if the file is in V-format and the LRECL of the file is less than the length of the record(s) being written. No message is issued and the return code is 0.

Example

The following program is an example of using the XEDIT interface to access files in storage.

```
XMPXFL   START
         USING *,R12
         LR    R12,R15                Establish addressability
         ST    R14,R14SAVE            Save return address

         USING FSCBD,R5               Mapping of FSCB
         LA    R5,XFSCB               Get FSCB address
         MVC   FSCBCOMM,=C'DMSXFLRD'  Put routine name in FSCB
         CMSCALL PLIST=(R5),          Read a line from XEDIT storage   X
               CALLTYP=X'02'
         LTR   R15,R15                If RC=0 then read was performed
         BNZ   ERRRD                  If RC¬=0, call an error routine

         ⋮

ERRRD    DS    0H                     Error routine here
         L     R14,R14SAVE            Get return address back
         BR    R14                    Return

XFSCB    FSCB  'ROCKFRD FILE A',      File ID to get info from
               FORM=E,                Need extended format FSCB
               BUFFER=MYBUFF,         Buffer to return info in
               BSIZE=MYLEN,           Length of buffer
               RECNO=1,               Access first record in ROCKFRD
               NOREC=1                Read one record in ROCKFRD
MYBUFF   DS    CL80                   Buffer to store info
MYLEN    EQU   80                     Buffer length

R14SAVE  DS    A                      Return address
         LTORG
         EJECT ,
         FSCBD                        Map FSCB
         REGEQU                       Define register equates
         END
```

Figure 21. Using the XEDIT Interface to Access Files in Storage

Chapter 10. Using the File System Macros

This chapter describes CMS macros you can use to manipulate CMS files, including:

- How you can use the file system macro for file I/O
- Sample programs that use file management macros.

File I/O Using FS Macros — A Typical Scenario

Here are suggested steps you may need to follow in order to open, read from, write to, and close a file:

1. Reserve space for and initialize a file system control block for the file — Before CMS can read from or write to a file, the file must have a file system control block (FSCB). You can use the FSCB macroinstruction to explicitly create a file system control block for files.

   If the storage for the FSCB is within the program, then you can specify the parameters on the FSCB macro to perform initialization of the FSCB. An example of a parameter for initialization would be the OPENTYP parameter. OPENTYP lets you indicate what type of open will be performed on the file.

   If the FSCB is within dynamically allocated storage (coded in a DSECT which maps that storage), the FSCB macro only reserves space within that DSECT and cannot be used to initialize the FSCB. Parameters for initialization need to be specified on the executable macros. In this case, the OPENTYP parameter would need to be specified on the FSOPEN macro.
2. Open a file — Use the FSOPEN macro to get a file ready for I/O. You can use FSOPEN to determine if the file exists and to check the attributes of the file. If you are opening a new file, you need to define the attributes of the file (such as file name or record format). If you plan to operate on an existing file, then information about the file is returned by FSOPEN.

   Although it is possible to open a file implicitly with a read or write, it is recommended that you use FSOPEN to open the file. FSOPEN lets you specify the intent of your operations plus check if the file has the correct characteristics for your operations.

   Note: The FSSTATE macro also lets you determine the status of a file. However, FSSTATE provides just a snap shot of the file for SFS. The state of the file could actually change before you perform an operation on it. FSOPEN holds an image of the file that matches the attributes of the file you will manipulate. FSOPEN places an implicit lock on the file. This lock protects the file from other programs or users, and retains any authority your program had to the file.
3. Perform the required operation — The FSREAD and FSWRITE macros allow you to perform sequential or nonsequential I/O to a file. You can implicitly open a file by using the FSREAD or FSWRITE without a previous FSOPEN. However, it is recommended to use FSOPEN (see note above).
4. Close a file — The FSCLOSE macro lets you close a file. Closing the file signifies that you are done with the type of operation you specified on the FSOPEN or FSCB macros (or implicitly identified by using FSREAD or FSWRITE). The file attributes are updated, but not necessarily written to disk. FSCLOSE releases the storage and resources that the file was using.

   Closing a file makes it available for other types of processing. For example, a file could be renamed or opened again for another operation. Closing the file does not necessarily commit any changes to disk or directory.

   Note: An error closing an SFS file may cause all updates on the work unit to be rolled back.
5. Commit changes made to file — For minidisk files, a commit is made when the last file opened for write is closed (using FSCLOSE) for that particular minidisk. For SFS files, a commit is made based on a work unit. The FS interface uses the default work unit that was current when the file was opened. When you use FSCLOSE on the last opened output file within a work unit, then FSCLOSE performs a coordinated commit on the work unit.

Note: An error during commit may cause all updates on the work unit to be rolled back.

6. Erasing CMS files — For minidisk or SFS files accessed as read/write, you can use FSERASE to delete a file.

Creating a File System Control Block

The file system control block (FSCB) is a PLIST that contains information about a CMS file. A recommended practice is to create or establish an FSCB for each file you wish to process. IBM recommends that you do not use the same FSCB to reference several different files. You can use multiple FSCBs to reference the same file. For example, you may want one FSCB for writing and a different FSCB for reading the file. File characteristics are inherent to the file and not to the FSCB.

If your application will be reentrant or run in a shared segment, then you will have to specify the address of an FSCB on each FS macro you invoke. The FSCB will need to be defined in a writeable storage area. Coding the FSCB macro only reserves storage, it does not initialize the FSCB. You can use the FSCBD macro to map to the FSCB and extract any information you need.

If your application will not be reentrant, you should still use an FSCB. It may suit your needs to have the FSCB be within your program. In this case, the FSCB is generated and initialized at compile time. In-line FSCBs make programs more efficient. You do not specify parameters (such as record number) on the operations (such as FSREAD or FSWRITE). You specify attributes of the operation with the FSCB macro. For example, the following macro placed in your program, creates and initializes a file system control block for the file INPUT TEST A1:

```
INFILE   FSCB  'INPUT TEST A1',OPENTYP=READ,FORM=E,BUFFER=BUFF,        *
                BSIZE=80,CACHE=YES
```

If you were to open a file specifying FSCB=INFILE on an FSOPEN, you do not need to specify the OPENTYP or CACHE parameters. These parameters would be already initialized by the FSCB macro. Likewise, by using FSCB=INFILE on an FSREAD, you would not need the BUFFER or BSIZE parameters on the FSREAD. Here is an example showing FSCB specified on executable macros. This coding method tends to use less space and can make programs efficient.

```
READ     FSREAD FSCB=INFILE,FORM=E,ERROR=INERR
         FSWRITE FSCB=OUTFILE,FORM=E,ERROR=OUTERR
         B     READ
         .
         .
         .
CLOSE    FSCLOSE FSCB=INFILE,ERROR=CLERR
         FSCLOSE FSCB=OUTFILE,ERROR=CLERR
         .
         .
         .
INFILE   FSCB   'INPUT FILE A1',BUFFER=SHARE,BSIZE=80,FORM=E
OUTFILE  FSCB   'OUTPUT FILE A1',BUFFER=SHARE,BSIZE=80,FORM=E
SHARE    DS    CL80
```

As a convenience, it is not necessary to specify FSCB on your executable FS macros. The FS macros will generate an in-line parameter list (FSCB) for that particular invocation. This technique can be less cumbersome and self-documenting, because you do not need to look at a corresponding FSCB macro to see which options are in effect. However, this method tends to require more space and tends to be less efficient than specifying FSCB on executable macros. Here is how the previous example would be coded without specifying the FSCB parameter:

```
READ      FSREAD 'INPUT FILE A1',BUFFER=BUFF,BSIZE=80,          *
                ERROR=RDERR,FORM=E
          FSWRITE 'OUTPUT FILE A1',BUFFER=BUFF,BSIZE=80,        *
                ERROR=WRTERR,FORM=E
          B     READ
          .
          .
          .
CLOSE     FSCLOSE 'INPUT FILE A1',ERROR=CLERR
```

```
          FSCLOSE 'OUTPUT FILE A1',ERROR=CLERR
BUFF      DS    CL80
```

Note that neither of these examples generate reentrant code. For an example of using FSCB macro in reentrant code, see the sample programs at the end of this chapter.

Contents of the File System Control Block

You can use the labels generated by the FSCBD DSECT to reference areas in the file system control block. The preferred practice though, is to use the FS macros to set these values. The FS macros provide for some error checking and handle any difference between extended and nonextended FSCB formats.

Note: You should not use explicit displacements to reference fields within the file system control block.

The FSCBD macro maps the file system control block as follows:

FSCBD DSECT Format:

```
*
         FSCBD
FSCBD    DSECT
FSCBCOMM DS    CL8            File system command (e.g. RDBUF)
FSCBFILE DS    CL18           File ID (name, type, and mode)
         ORG   FSCBFILE
FSCBFNFT DS    CL16           File name and file type
         ORG   FSCBFNFT
FSCBFN   DS    CL8            File name
FSCBFT   DS    CL8            File type
FSCBFM   DS    CL2            File mode (letter and number)
         ORG   FSCBFM
FSCBFML  DS    CL1            File mode letter
FSCBFMN  DS    CL1            File mode number
FSCBITNO DS    H              Relative record number to be
                              accessed on FSREAD and FSWRITE
                              (applies only to the nonextended
                              FSCB)
FSCBBUFF DS    A              Address of the input/output buffer
                              for FSREAD and FSWRITE (also used
                              on calls to DMSSTT for the FST
                              address)
FSCBSIZE DS    F              Length (in bytes) of the input/output
                              buffer (also used to return the record
                              length on FSOPEN)
FSCBFV   DS    CL2            Record format and first flag byte
         ORG   FSCBFV
FSCBRECF DS    CL1            Record format - F or V
FSCBFLG  DS    XL1            First flag byte
*
*        'FSCBFLG' flag byte definition
*
FSCBTHEX EQU   X'80'          Space threshold exceeded (SFS only)
FSCBITAV EQU   X'40'          Item available (no longer used)
FSCBEPL  EQU   X'20'          Extended PLIST (FORM=E)
FSCBMSG  EQU   X'10'          MSG=YES on FSSTATE or FSOPEN
FSCBSTW  EQU   X'08'          STATEW specified on FSSTATE
FSCBCACY EQU   X'04'          CACHE=YES specified
FSCBCACN EQU   X'02'          CACHE=NO specified
FSCBRCAV EQU   X'01'          Previous record null (no longer used)
FSCBNOIT DS    H              Number of records to be accessed on
                              FSREAD and FSWRITE (applies only to the
                              nonextended FSCB)
         ORG   FSCBNOIT       Extended format fields defined
                              over nonextended FSCBNOIT
FSCBFLG2 DS    XL1            Second flag byte
*
*        'FSCBFLG2' flag byte definition (FORM=E only)
*
FSCBNMAC EQU   X'80'          NOMSG=ACTIVE specified on FSOPEN
FSCBNMNF EQU   X'40'          NOMSG=NOTFOUND specified on FSOPEN
FSCBNMOS EQU   X'20'          NOMSG=OSDOS specified on FSOPEN
FSCBOTYP DS    CL1            OPENTYP value
*
*        'FSCBOTYP' Values (FORM=E only)
*
FSCBTNON EQU   X'00'          OPENTYP=NONE specified
FSCBTRD  EQU   C'R'           OPENTYP=READ specified
```

```
FSCBTWR  EQU   C'W'           OPENTYP=WRITE specified
FSCBTNEW EQU   C'N'           OPENTYP=NEW specified
FSCBTREP EQU   C'X'           OPENTYP=REPLACE specified
FSCBNORD DS    F              Number of bytes actually
                              read on FSREAD
             ORG   FSCBNORD
*
*  'FSCBFST' is returned on FSOPEN.  Its value
*  is based on the OPENTYP specified and whether or not
*  the file exists. Note that a nonextended format FSCB
*  (FORM=E not specified) implies OPENTYP=NONE.  The values
*  are as follows:
*
*  File doesn't exist .... FSCBFST=A(0)
*
*  File exists:
*  Not FORM=E ............ FSCBFST=A(Copy of 40 byte FST)
*  OPENTYP=NONE .......... FSCBFST=A(Copy of 64 byte FST)
*  OPENTYP=READ .......... FSCBFST=A(Copy of 64 byte FST)
*  OPENTYP=WRITE ......... FSCBFST=A(Copy of 64 byte FST)
*  OPENTYP=REPLACE ....... FSCBFST=A(-1)
*  OPENTYP=NEW ........... Error, FSCBFST is unchanged
*
FSCBFST  DS    A              Address of a copy of the FST
                              returned on FSOPEN
*
* The following fields apply only to the extended form FSCB
* (i.e., FORM=E was specified).
*
FSCBAITN DS    F              Relative record number to be
                              accessed on FSREAD and FSWRITE
                              (also referred to as
                              the "alternate item number")
FSCBANIT DS    F              Number of records to be accessed
                              on RSREAD and FSWRITE (also
                              referred to as the "alternate
                              number of items")
FSCBWPTR DS    F              Extended write pointer (input
                              on FSPOINT FORM=E, output on
                              FSOPEN)
FSCBRPTR DS    F              Extended read pointer (input
                              on FSPOINT FORM=E, output on
                              FSOPEN)
FSCBLNBY EQU   *-FSCBD        Length (in bytes) of the extended
                              FSCB
```

The FSCBAITN, FSCBANIT, FSCBWPTR, and FSCBRPTR fields are generated in the FSCB only when the extended format (FORM=E) is specified. If FORM=E is specified, the FSCBITNO and FSCBNOIT fields are reserved for other purposes.

Mapping the File System Control Block

Regardless of whether you use the FSCB macro to create an FSCB explicitly, or you use the FSREAD or FSWRITE macros to create an FSCB implicitly, use the FSCBD macro to map the file system control block.

Using FSCBD DSECT Labels

You can also use the labels generated by the FSCBD DSECT to reference, examine, and change fields in the file system control block explicitly.

For example, the following code opens a file and checks the record length and record format of the file:

```
                 LA    R5,INFSCB
                 USING FSCBD,R5
                 FSOPEN FSCB=(R5),FORM=E
                 CLC   FSCBSIZE,=AL4(L'BUFF)
                 BNE   BADRECL
                 CLI   FSCBRECF,C'F'
                 BNE   NOTFIXED
                 .
                 .
        INFSCB   FSCB  'INPUT TEST A1',
                       BUFFER=BUFF,
                       BSIZE=80,
                       OPENTYP=READ,
                       FORM=E,
                       RECFM=Ffile system control block:
```

The following code changes the file name in a file system control block:

```
                       LA    R5,INFSCB
                       USING FSCBD,R5
                       .
                       .
                      MVC   FSCBFN,NEWNAME
                      .
                      .
            INFSCB   FSCB  'INPUT TEST A1',FORM=E
            NEWNAME  DC    CL8'OUTPUT'
                      FSCBD
```

Modifying Fields in the File System Control Block

There are several options you can specify on the FSCB, FSREAD, FSWRITE, and FSOPEN macros that allow you to specify or change the values in the file system control block.

The options you can specify include:

- RECFM — Whether the file record format is fixed or variable.
- BUFFER — The address of a buffer from which records are to be read or written. You must specify a BUFFER value before you perform a read or write operation.
- FORM — Whether an extended format file system control block is to be generated. An extended format file system control block allows you to specify a value (up to 2^{31}-1) for RECNO and NOREC. If you do not specify FORM=E, the RECNO and NOREC values cannot exceed 65535. Users should always code FORM=E on the FSCB, FSOPEN, FSREAD, FSWRITE, FSPOINT, and FSSTATE.

  Whenever you use the FSCB parameter on any of the executable FS macros, the FORM parameter must be the same on the executable macro and the FSCB it references.
- BSIZE — The number of bytes to be read or written for each read or write request.
- RECNO — The record number of the next record to be accessed, relative to the beginning of the file, record 1. The value 0 indicates that records are to be accessed sequentially.
- NOREC — The number of records to be accessed in the next operation. This value must be 1 for V-format files.

Using the File System Control Block

The following example shows you how to code an FSCB macroinstruction to define various file and buffer characteristics and how to use the same file system control block to refer to different files:

```
         FSREAD 'INPUT FILE A1',FSCB=COMMON,FORM=E
         FSWRITE 'OUTPUT FILE A1',FSCB=COMMON,FORM=E
         .
         .
         .
COMMON   FSCB  BUFFER=SHARE,RECFM=V,BSIZE=200,FORM=E
SHARE    DS    CL200
```

In the above example, the fileid specifications on the FSREAD and FSWRITE macroinstructions modify the file system control block at the label COMMON each time a read or write operation is performed .

IBM recommends however, that you do not use the same FSCB to reference several different files. If you must, you can override the *fileid* and any of the other options on the FSOPEN, FSWRITE, or FSREAD macroinstructions when you reference a file by way of its FSCB. If, however, you use the FSOPEN macro to open an existing file, CMS resets the BSIZE and RECFM fields in the FSCB to reflect actual file characteristics, not necessarily the characteristics you specify on FSOPEN.

When you use the same FSCB for multiple files, care must be taken to specify the appropriate FSCB options on any other macros that reference the FSCB, particularly when the options differ from file to file. Each time these options are specified on another macro (FSOPEN, FSREAD, FSWRITE) the FSCB is modified. This may lead to an error if a subsequent operation for a different file is issued which allows an option to default to the value present in the FSCB.

For example:

```
             FSWRITE 'NEW FILE A1',FSCB=OUTFSCB,RECFM=F,FORM=E
             FSWRITE 'OLD FILE A1',FSCB=OUTFSCB,FORM=E
             .
             .
             .
OUTFSCB   FSCB    RECFM=V,BUFFER=RECAREA,BSIZE=80,FORM=E
```

Even though OUTFSCB has RECFM=V specified, the FSWRITE to 'NEW FILE A1' with RECFM=F will change OUTFSCB to contain RECFM=F. The second FSWRITE to 'OLD FILE A1' will assume RECFM=F (not V) because that is the value that is now in the FSCB. Thus, if the file 'OLD FILE A1' on disk is actually RECFM=V, an error will occur on the second FSWRITE, even though the FSCB had specified RECFM=V. To avoid this problem, the preferable approach is to code a separate FSCB for each file which is being used. Otherwise, you must specify the option (in this example, RECFM) on each FSWRITE, FSREAD, and so on, which references the same FSCB.

Opening CMS Files

Use the FSOPEN macro to establish a logical connection to a file for subsequent reads or writes or both. The FSOPEN macro allows you to open a file and specify the intent, or the type of operation that you will be performing, and the type of I/O that you want performed.

You can indicate the intent by specifying one of the following values for the OPENTYP parameter:

READ means that the file will only be read. You cannot open a file for READ if it does not exist.

WRITE indicates that the file may be written to or read from. All changed and added records are written. The other records you do not modify will remain unchanged. If the file does not exist, it is created.

NEW indicates that the file does not exist and it will be created. It may then be read from or written to. If the file already exists, it is an error and the file is not opened.

REPLACE means that the file is replaced with only the subsequently written records. If the file does not exist, it is created. You can only read records that you have written (otherwise you will receive an end-of-file condition). If you close the file without writing any records, the contents of the file are unchanged.

NONE indicates that you do not intend to open the file at this time. This is essentially equivalent to an FSSTATE, and differs from an FSSTATE in that it may be used to create an FSCB for the file.

Note that you must specify FORM=E to use the OPENTYP parameter.

When you open a file, CMS will update the fields of the file system control block and for existing files will return the address of a copy of the file's FST. Fields in the FSCB and FST will reflect the actual characteristics of the file. Information returned in the FSCB and FST together include the number of records in the file, record length, record format, read and write pointers, and date and time the file was last modified. You can use the FSCBD macro to access fields within the FSCB. You can use the FSTD macro to access fields in the copy of the FST.

If an existing F-format file is opened with an intent of WRITE, the length of each record is determined at open from the file's directory entry and cannot be changed by the first write to the file. If an existing F-format file is opened with an intent of REPLACE, the length of each record is the length of the first record written to the file, just as it would be if the file did not exist.

You can also use the FSOPEN (and consequently the FSCB) macro to indicate whether caching of multiple data blocks is to be performed for this file. To use the CACHE parameter, FORM=E must also be specified. If FORM=E is not specified, or the file is not explicitly opened, then CACHE=DEFAULT is assumed when the file is opened by way of the first FSREAD, FSWRITE, or FSPOINT. See "Caching" on page 119 for some suggestions on caching.

Reading and Writing CMS Files

This section contains several examples that illustrate how to use the FSREAD and FSWRITE macros to read and write files.

Single Reads and Writes

By default, FSREAD and FSWRITE access files sequentially. When you read files with the FSREAD macroinstruction, reading begins with record number 1. When you write records to an existing file with the FSWRITE macro, writing begins following the last record in the file.

The following example shows how to use FSREAD and FSWRITE to read records from one file and write them into another:

```
         FSREAD FSCB=INFILE,FORM=E,ERROR=INERR
         FSWRITE FSCB=OUTFILE,FORM=E,ERROR=OUTERR
         .
         .
         .
INFILE   FSCB   'INPUT FILE A1',BUFFER=SHARE,BSIZE=80,FORM=E
OUTFILE  FSCB   'OUTPUT FILE A1',BUFFER=SHARE,BSIZE=80,FORM=E
SHARE    DS    CL80
```

Multiple Reads and Writes to a Fixed File

By default, FSREAD and FSWRITE operate on one record at a time. The following example shows how you can use the NOREC parameter to read and write more than one fixed-length record at a time:

```
         FSREAD FSCB=INFILE,NOREC=10,FORM=E,ERROR=INERR
         FSWRITE FSCB=OUTFILE,NOREC=10,FORM=E,ERROR=OUTERR
         .
         .
         .
INFILE   FSCB   'INPUT FILE A1',BUFFER=SHARE,BSIZE=10*80,FORM=E
OUTFILE  FSCB   'OUTPUT FILE A1',BUFFER=SHARE,BSIZE=10*80,FORM=E
SHARE    DS    CL(10*80)
```

Records in a file with variable-length records can ONLY be read or written one at a time.

Variable Length Records

When you write variable-length records, you must specify RECFM=V either in the file system control block for the file or on the FSWRITE or FSREAD macroinstruction. The read/write buffer should be large enough to accommodate the largest record you read or write.

When you write variable-length records, use the BSIZE= parameter on the FSWRITE macroinstruction to indicate the record length for each record you write.

The following example shows how you could read and write a variable-length file. Note, on return from FSREAD, register 0 contains the actual number of bytes read.

```
READ   FSREAD 'DATA CHECK A1',BUFFER=SHARE,BSIZE=130,             *
               ERROR=OUT,FORM=E
         FSWRITE 'COPY DATA A1',BUFFER=SHARE,BSIZE=(R0),          *
               RECFM=V,FORM=E
         B     READ
         .
         .
         .
SHARE   DS    CL130
```

Note: When you update files of variable-length records, the replacement record must be the same length as the original record. An attempt to write a record longer or shorter than the original record results in truncation of the file at the specified record number. No error return code is given.

Reading Specific Records

CMS uses pointers to keep track of which records were last written and read. To read or write a specific record, you can specify the RECNO parameter of the FSREAD or FSWRITE macros. You can also use the FSPOINT macro to reset these pointers. For example, the following code illustrates how you could read the 10th and then the 25th records in a file:

```
          FSREAD FSCB=RFSCB,RECNO=10,FORM=E
          .
          .
          .
          FSREAD FSCB=RFSCB,RECNO=25,FORM=E
          .
          .
          .
RFSCB     FSCB 'INPUT FILE A1',BUFFER=COMMON,BSIZE=120,         *
               FORM=E,RECFM=F
COMMON    DS    CL120
```

The following code illustrates how you could read 5 records, skip to record 10, and read 5 more records.

```
        .
        .
        .
        LA      R5,5
READLP1 FSREAD  FSCB=RFSCB,FORM=E
        .
        .
        .
        BCT     R5,READLP1
*
        FSPOINT FSCB=RFSCB,RDPNT=10,FORM=E
*
        LA      R5,5
READLP2 FSREAD  FSCB=RFSCB,FORM=E
        .
        .
        .
        BCT     R5,READLP2
        .
        .
        .
RFSCB   FSCB    'INPUT FILE A1',BUFFER=COMMON,BSIZE=120,        *
                FORM=E,RECFM=F
COMMON  DC      CL120
```

End-of-File Checking

When CMS reaches the end of a file, it returns in register 15 a return code of 12. You can use the ERROR= operand of the FSREAD macro to specify an error handling routine to check for end of file.

End-of-file occurs on the first read where RECNO is beyond the end of a file. When RECNO=0 or records are being read sequentially, end-of-file occurs when the read pointer is greater than the last record number of file.

Monitoring SFS Filespace Threshold

The FSCBTHEX (X'80') indicator bit of the FSCBFLG byte indicates when you have reached your SFS filespace threshold. Because the CMS portion of the file system does buffering, you will only see the indicator when it is necessary to write the buffers to the file pool. This can occur during a read, write, or close. For small files, the indicator might not be returned until the close.

If your application is using non-shared SFS files, (see "Using Programs on Non-Shared SFS Files" on page 119), the file system will return a return code of 13 in register 15 when you have reached your file space limit. If your application is sharing files, you must examine the threshold indicator to determine whether you have enough blocks available to continue writing to the file space.

Closing Files and Committing Changes

When a program completes (end-of-command), CMS closes all files that have been left open and implicitly commits any changes. However, it is recommended that you close (with FSCLOSE) all files you have opened (with FSOPEN, FSREAD, FSWRITE or FSPOINT). You need to close all output files on a minidisk to commit any changes. Note that the CMS FINIS command is equivalent to FSCLOSE.

Closing a file makes it available for other types of processing. For example, a file could be renamed or opened again for another operation. Closing the file does not necessarily commit any changes to disk. However, all files accessed with the FS interface need to be closed before any commit can take place.

For minidisk files, a commit is made when the last file opened for write is closed (using FSCLOSE) for that particular minidisk. For SFS files, a commit is made based on a work unit. The FS interface uses the default work unit that was current when the file was opened. When you use FSCLOSE on the last file that was opened (or the last output file opened) within a work unit, then FSCLOSE performs a coordinated commit.

Note: An error closing an SFS file may cause all updates on the work unit to be rolled back.

Note For EXEC Writers

Suppose, you call three programs, one after another, from within an exec, and the programs do not commit work. The *end of command* is after all three programs execute and the exec ends, not after each program ends. In this case, changes are not committed until all three programs successfully run and the exec ends.

The problem is that you cannot always predict how users will use your program. While you may want it to run stand-alone, there is no guarantee that a user would not put it in an exec with other programs, or call it using some other mechanism.

Erasing Files

Use the FSERASE macroinstruction to delete a CMS disk file. For example, to delete the file GETRID OFITNOW A1, code

```
         FSERASE FSCB=TEMP
         .
         .
         .
TEMP     FSCB 'GETRID OFITNOW A1',FORM=E
```

Note: To use FSERASE on a minidisk file, the user must have accessed the minidisk read/write. To erase a shared file, the user must have accessed the directory read/write and have read/write authority to the file. If the file is an SFS alias, only the alias is erased. The base file remains intact. When the file is an SFS base file, all authorities and aliases to that file are dropped.

Sample Programs

The following programs illustrate the use of the CMS file management macros. The first program illustrates the nonreentrant coding technique. The second program illustrates the reentrant coding technique.

Nonreentrant

This program illustrates the nonreentrant coding technique. Nonreentrant means that the program either modifies its external program parameters or its own instructions/storage areas during execution.

A SAMPLE FILE MANAGEMENT PROGRAM - NONREENTRANT

```
* Function:
* This program makes a copy of an existing file under another
* name; if a file of that name already exists, it is replaced
* by the copy. The existing file must have fixed-length
* 80-byte records.
* Example invocation:
*    COPYF80 INPUT FILE A OUTPUT FILE A
* Input:
*    Tokenized PLIST in R1
* Output:
*    Either the expected copy of the file or an error message.
*    The return code is zero if the copy was successful and
*    nonzero otherwise.
* Note:
*    This version illustrates the nonreentrant coding technique.
COPYF80  CSECT
       LR    R12,R15          establish code addressability
       USING COPYF80,R12      using the entry point address
       USING PLIST,R1         passed in R15
                              R1 -> tokenized PLIST
* The address of the parameter list is in register 1.  It
* contains seven doubleword fields:  the first field contains
* the name of the command that invokes this program; the
* remaining six fields contain the file names, file types,
* and file modes of the input and output files.  The file
* ID of the input file starts 8 bytes from register 1, the
* file ID of the output file starts 32 bytes away.
*
       LA    R2,INFILE        R2 = address of input file ID
                              in PLIST
       LA    R3,OUTFILE       R3 = address of output file ID
                              in PLIST
       DROP  R1
* Open the input file.  Specifying the file id in a register
* will cause it to be copied into the FSCB by the FSOPEN macro
* expansion. Let FSOPEN handle any errors:
*
           FSOPEN (R2),             input file ID                 *
                 FSCB=INFSCB,       FSCB for input file           *
                 OPENTYP=READ,     opening for READ              *
                 MSG=YES,          let FSOPEN issue error
                                   messages                      *
                 ERROR=INERR,      where to go on an error       *
                 FORM=E            extended format FSCB
* Verify that the input file has fixed 80-byte records.
       LA    R10,INVFILE      set error return code
       USING FSCBD,R1
       CLI   FSCBRECF,C'F'    input file have fixed-length
                              records?
       BE    CHKLRECL         branch if yes
       APPLMSG APPLID=CMS,    tell user if no                    *
             TEXT='Input file does not have fixed-length
                   records'
       B     CLOSEIN          close the open input file and
                              exit
CHKLRECL DS    0H
       CLC   FSCBSIZE,=F'80'  input file record length = 80?
       BE    OPENOUT          branch if yes
       APPLMSG APPLID=CMS,    tell user if no                    *
             TEXT='Input file does not have 80-byte records'
       B     CLOSEIN          close the open input file and exit
       DROP  R1
* Open the output file.  Let FSOPEN handle any errors:
OPENOUT  DS    0H
       FSOPEN (R3),           output file ID                     *
             FSCB=OUTFSCB,    FSCB for output file               *
             OPENTYP=REPLACE, opening for REPLACE                *
             MSG=YES,         let FSOPEN issue error messages    *
             ERROR=CLOSEIN,   close input file and exit          *
             FORM=E           extended format FSCB
* Both files are now open.  Read a record from the input file
* and write the record on the output file.
READLOOP DS    0H               read in and write out a record
       FSREAD FSCB=INFSCB,    input file FSCB updated by FSOPEN *
             ERROR=EOF,       where to go on a read error        *
             FORM=E           extended format FSCB
       FSWRITE FSCB=OUTFSCB,  output file FSCB                   *
             ERROR=WRITERR,   where to go on a write error       *
             FORM=E           extended format FSCB
       B     READLOOP         loop back for next record
* Come here when a read of the input file failed.
EOF    DS    0H
       C     R15,=A(ENDFILE)  is failure end-of-file?
       BNE   READERR          error if not end-of-file
       SR    R10,R10          otherwise, normal completion
       APPLMSG APPLID=CMS,    tell user                         *
             TEXT='Copying is complete'
       B     CLOSEOUT         close files and exit
*
* Error routine if an error occurred while writing a file:
WRITERR  DS    0H
         LR    R10,R15          save return code in R10
         APPLMSG APPLID=CMS,    tell user                       *
            TEXT='Error code &&1 writing file',  *
               SUB=(DEC,(R10))
         C     R10,=A(ROLLBACK)
         BE    EXIT
         B     CLOSEOUT         close files and exit

* Control reaches this point if the input file failed to open.
INERR    DS    0H
         LR    R10,R15          save FSOPEN return code in R10
         B     EXIT             exit to the caller
* Control reaches this point if a read error was not end of file.
READERR  DS    0H
         LR    R10,R15          save return code in R10
         APPLMSG APPLID=CMS,    tell user                        *
               TEXT='Error code &&1 reading file',*
               SUB=(DEC,(R10))
         C     R10,=A(ROLLBACK)
         BE    EXIT
CLOSEOUT DS    0H
         FSCLOSE FSCB=OUTFSCB   close the output file
         LTR   R15,R15          successful?
         BZ    CLOSEIN          yes - go close input
         LR    R10,R15          no - save return code
         C     R15,=A(ROLLBACK) did an SFS rollback occur?
         BE    EXIT             yes - no need to close input
                                file
CLOSEIN  DS    0H
         FSCLOSE FSCB=INFSCB    close the input file
         LTR   R15,R15          successful?
         BZ    EXIT             yes - cleanup and exit
         LR    R10,R15          no - save return code
* Exit
EXIT     DS    0H
         CMSRET RC=(R10)        return to caller
*
* Working storage:
INFSCB   FSCB  BUFFER=BUFF,BSIZE=80,FORM=E,CACHE=YES
OUTFSCB  FSCB  BUFFER=BUFF,BSIZE=80,FORM=E,CACHE=YES
BUFF     DS    CL80             buffer for records
* Return codes:
INVFILE  EQU   24               invalid file return code
ENDFILE  EQU   12               end-of-file return code
ROLLBACK EQU   31               SFS rollback return code
* DSECT to define the tokenized input PLIST:
PLIST    DSECT                  input tokenized PLIST
         DS    CL8              command used to invoke the
                                program
INFILE   DS    3CL8             file name, type, mode of input
                                file
OUTFILE  DS    3CL8             file name, type, mode of output
                                file
         REGEQU                 register equates
         FSCBD                  FSCB mapping
         END
```

Reentrant

This program illustrates the reentrant coding technique. This program can be entered repeatedly and can be entered before prior executions are complete. For reentrant programs, neither their external program parameters nor instructions can be modified during execution.

A SAMPLE FILE MANAGEMENT PROGRAM - REENTRANT

```
* Function:
*  This program makes a copy of an existing file under another
*  name; if a file of that name already exists, it is replaced
*  by the copy.
*  The existing file must have fixed-length 80-byte records.
* Example invocation:
*    COPYF80 INPUT FILE A OUTPUT FILE A
* Input:
*    Tokenized PLIST in R1
* Output:
*    Either the expected copy of the file or an error message.
*    The return code is zero if the copy was successful and
*    nonzero otherwise.
* Note:
*    This version illustrates the reentrant coding technique.
*
*
COPYF80  CSECT
         LR    R12,R15      establish code addressability using
         USING COPYF80,R12  the entry point address passed
                            in R15
         USING PLIST,R1     R1 -> tokenized PLIST
* The address of the parameter list is in register 1.  It
* contains seven doubleword fields:  the first field contains
* the name of the command that invokes this program; the
* remaining six fields contain the file names, file types,
* and file modes of the input and output files.  The file ID
* of the input file starts 8 bytes from register 1, the file ID
* of the output file starts 32 bytes away.
        LA    R2,INFILE     R2 = address of input file ID in
                            PLIST
        LA    R3,OUTFILE    R3 = address of output file ID in
                            PLIST
        DROP  R1
* Get local working storage; abend with a message if no
* storage is available.
        LA   R0,WKGSTGSZ      specify size needed
        CMSSTOR OBTAIN,       request the storage            *
              BYTES=(R0),                                    *
              ERROR='ABEND',                                 *
              MSG=YES
        LR    R13,R1          copy obtained storage address
                              to R13
        USING WKGSTG,R13      make working storage addressable

* To initialize the FSCB efficiently,we copy an FSCB from within
* the module which was initialized at assemble time with all the
* parameters we want that can be specified on the FSCB. Thus,
* OPENTYP, RECNO, NOREC, CACHE are all specified on the static,
* unmodified FSCB.  When this FSCB is copied to the one in the
* working storage, the target FSCB will thus be initialized.
* The parameters mentioned above will not need to be specified
* on the executable macros, such as FSOPEN and FSREAD, and the
* macros will generate more efficient code since they will not be
* filling in these constant values in the FSCB.
*
       MVC   INFSCB,INFSCBI   Initialize the FSCB in working
                              storage
*
* Open the input file. Specifying the file id in a register
* will cause it to be copied into the FSCB by the FSOPEN
* macro expansion. The BUFFER parameter is specified here
* once (instead of on each read) for efficiency, since it
* does not change on the reads; FSOPEN will set BSIZE and
* RECFM to the record length and format of the existing
* file. Let FSOPEN handle any errors.
*
       LA    R4,BUFF          get address of record buffer
       FSOPEN (R2),           input file ID                  *
           FSCB=INFSCB,     FSCB for input file              *
           ERROR=INERR,     where to go on an error          *
           MSG=YES,         let FSOPEN issue error messages  *
           BUFFER=(R4),     where to read the records into   *
           FORM=E           extended format FSCB
*
* Verify that the input file has fixed eighty-byte records.
*
       LA    R10,INVFILE    set error return code
       USING FSCBD,R1
       CLI   FSCBRECF,C'F'  input file have fixed-length
                            records?
       BE    CHKLRECL       branch if yes
       APPLMSG APPLID=CMS,  tell user if no                  *
             TEXT='Input file does not have fixed-length
                   records'
       B     CLOSEIN        close the open input file and
                            exit
CHKLRECL DS    0H
       CLC   FSCBSIZE,=F'80'  input file record length = 80?
       BE    OPENOUT          branch if yes
       APPLMSG APPLID=CMS,    tell user if no                *
             TEXT=&apos.Input file does not have 80-byte
                  records&apos.'
       B     CLOSEIN          close the open input file and
                              exit
       DROP  R1

* Open the output file.  The BUFFER, BSIZE, RECNO, and NOREC
* parameters are specified here once (instead of on each write)
* for efficiency, since they do not change on the writes.  To
* ensure OUTFSCB gets properly initialized, we first clear it to
* all binary zeros, then specify all FSOPEN parameters.  Since
* specifying FSCB on FSOPEN means there are no default values for
* those parameters which are available on the FSCB macro, we must
* specify them here so that the fields within the FSCB get
* initialized; otherwise, fields like RECFM might contain
* invalid values.
* * (Note: This does not produce as efficient code as the
*        technique used to initialize and open INFSCB, but is
*        perhaps more understandable, since the reader can
*        clearly see all the parameters in effect on this
*        FSOPEN without having to look at the FSCB).
OPENOUT  DS    0H
         XC    OUTFSCB,OUTFSCB  clear the FSCB in working
                                torage
         FSOPEN (R3),      output file ID                    *
         FSCB=OUTFSCB,     FSCB for output file              *
         OPENTYP=REPLACE,  REPLACE existing file or make
                           new one                           *
         CACHE=YES,        use multiple file system buffers  *
         MSG=YES,          let FSOPEN issue error messages   *
         NOMSG=,           no special error message
                           suppression                       *
         ERROR=CLOSEIN,    where to go on an error           *
         RECFM=F,          format of the records
                           to be written                     *
         BSIZE=80,         length of the records
                           to be written                     *
         BUFFER=(R4),      where to write the records from   *
         RECNO=0,          use strictly sequential access    *
         NOREC=1,          number of records to write
                           per FSWRITE                       *
         FORM=E            extended format FSCB
*
* Both files are now open.  Read a record from the input
* file and write the record on the output file.
*
READLOOP DS    0H               read in and write
                                out a record
         FSREAD FSCB=INFSCB,    input file FSCB updated
                                by FSOPEN                    *
               ERROR=EOF,       where to go on a read error  *
               FORM=E           extended format FSCB
         FSWRITE FSCB=OUTFSCB,  output file FSCB             *
               ERROR=WRITERR,   where to go on a write error *
               FORM=E           extended format FSCB
         B     READLOOP         loop back for next record

* Come here when a read of the input file failed.
EOF      DS    0H
         C     R15,=A(ENDFILE)  is failure end-of-file?
         BNE   READERR          error if not end-of-file
         SR    R10,R10          otherwise, normal completion
         APPLMSG APPLID=CMS,    tell user                       *
               TEXT='Copying is complete'
         B     CLOSEOUT         close files and exit
*
* Error routine if an error occurred while writing a file:
WRITERR DS    0H
         LR    R10,R15          save return code in R10
         APPLMSG APPLID=CMS,    tell user                       *
```

```
               TEXT='Error code &&1 writing file',      *
               SUB=(DEC,(R10))
         C     R10,=A(ROLLBACK)
         BE    EXIT
         B     CLOSEOUT     close files and exit
*
* Control reaches this point if the input file failed to open.
INERR    DS    0H
         LR    R10,R15          save FSOPEN return code in R10
         B     EXIT             exit to the caller
*
* Control reaches this point if a read error was
not end of file.
READERR  DS    0H
         LR    R10,R15          save return code in R10
         APPLMSG APPLID=CMS,    tell user                      *
               TEXT='Error code &&1 reading file',   *
               SUB=(DEC,(R10))
         C     R10,=A(ROLLBACK)
         BE    EXIT
CLOSEOUT DS    0H
         FSCLOSE FSCB=OUTFSCB   close the output file
         LTR   R15,R15          successful?
         BZ    CLOSEIN          yes - go close input
         LR    R10,R15          no - save return code
         C     R15,=A(ROLLBACK) did an SFS rollback occur?
         BE    EXIT             yes - no need to close
                                input file
CLOSEIN  DS    0H
         FSCLOSE FSCB=INFSCB    close the input file
         LTR   R15,R15          successful?
         BZ    EXIT             yes - cleanup and exit
         LR    R10,R15          no - save return code

* Release the working storage and return to the caller.
EXIT     DS    0H
         LA    R0,WKGSTGSZ      get size of storage
                                to release
         LR    R2,R13           copy address of storage
                                to release
         CMSSTOR RELEASE,       release the storage            *
               ADDR=(R2),       location of the storage
                                to release                     *
               BYTES=(R0)       size of the storage
                                to release
         CMSRET RC=(R10)        return to caller
*
* Constants: *
* INFSCBI is an FSCB used only as a skeleton to
* initialize INFSCB.  It cannot be used directly on
* executable FS macros, since they modify the FSCB;
* since INFSCBI resid  es within this program, it
* must not be modified since this module is re-entrant.
*
INFSCBI   FSCB  FORM=E,    extended format FSCB                *
          OPENTYP=READ,    provide READ access only            *
          CACHE=YES,       use multiple file system buffers  *
          RECNO=0,         use strictly sequential access    *
          NOREC=1          number of records to read
                           per FSREAD
*
*Return codes:
INVFILE  EQU   24          invalid file return code
ENDFILE  EQU   12          end-of-file return code
ROLLBACK EQU   31          SFS rollback return code
*
* Working storage:
WKGSTG   DSECT INFSCB
         FSCB  FORM=E      input FSCB
OUTFSCB  FSCB  FORM=E      output FSCB
BUFF     DS    CL80        buffer for records
         DS    0D          align to doubleword boundary
WKGSTGSZ EQU   *-WKGSTG    working storage size
*
* DSECT to define the tokenized input PLIST:
PLIST
         DSECT             input tokenized PLIST
         DS    CL8         command used to invoke the program
INFILE   DS    3CL8        file name, type, mode of input file
OUTFILE  DS    3CL8        file name, type, mode of output file
         REGEQU            register equates
         FSCBD                FSCB mapping
         END
```

Chapter 11. Unit Record Devices and Tapes

This chapter describes:

- How to use CMS macros to perform unit record I/O (print on printers, punch on punches, and read from readers). This includes descriptions of the following macros:
  - PRINTL — Prints a line on the printer
  - CMSDEV — Provides device information for PRINTL
  - PUNCHC — Punches a card
  - RDCARD — Reads a record from the reader.
- The macros you can use to manage tapes.
- How to process tape labels in CMS.
- How to use Tape Library Dataservers under CMS OS Simulation.

Printing

Use the PRINTL macroinstruction to write a line or multiple lines to the virtual printer at virtual address X'000E'. PRINTL lets you specify:

- The data to be printed (this can be actual text, the address of the text, the address of a buffer (for fixed-length records), or the address of a list that contains addresses of lines to be printed.
- Carriage control characters to specify how many lines should be skipped before the next line is printed.
- Table reference character (TRC) bytes to select the 3800 translate table to be used.
- The address of a 12-byte storage area to contain the device characteristics provided by the CMSDEV macro.
- The address of an error routine and various macro formats.

For the complete syntax of the PRINTL macro see *z/VM: CMS Macros and Functions Reference*.

Determining How Many Bytes You Can Print

The following table lists the maximum number of data bytes you can print for the various printers:

Table 20. Maximum Data Bytes You Can Print

Virtual Printer Type	Maximum Data Bytes
1403	132
3203	132
3800	204
4248	168
VAFP (Virtual Advanced Function Printer)	32767

To determine the line length, add the following to your bytes of data:

- One byte for the carriage control character if CC=YES is specified,
- One byte for the TRC byte if TRC=YES is specified.

If you do not specify the length, it defaults to 133 characters, unless ‘linetext’ is specified. In this case, the length is taken from the length of the line text.

Lines which are greater than the carriage size will not be printed and a return code of 1 will be issued. However, lines with a carriage control character of X'5A' may have lengths up to 32767 bytes. If you use quoted data with a X'5A' carriage control, the line length must not be greater than 256 bytes.

Using PRINTL

Before you use PRINTL, you need to know that PRINTL does not print lines on a real printer—PRINTL prints lines in a file, called a spool file, which is kept in a queue, called a spool file queue, that is associated with a real device. Because you can assign device attributes to the spool file queue or, for that matter, to specific spool files, you are said to have a virtual printer.

Without going into detail about the concept of virtual unit record devices it is important to note that you can use CP and CMS commands to control your virtual printer. For example, you can use the CP LOADVFCB and SPOOL commands to define the forms control buffer image to be used; you can use the CMS SETPRT command if you use a virtual 3800; and you must use the CP CLOSE command to close your printer file before CMS transfers it to CP for printing on a real device.

For more information on unit record devices, see the *z/VM: CMS User's Guide*.

Carriage Control Characters

Carriage control characters let you control the vertical spacing when you print. You can use the CC parameter of the PRINTL macro to specify a carriage control character. If you do not specify one, CMS spaces one line before printing the next line.

The carriage control character may be either ASA (ANSI) or machine code. The valid ASA control characters are:

```
Character  Hex Code    Meaning

   blank      40       Space 1 line before printing
   0          F0       Space 2 lines before printing
   -          60       Space 3 lines before printing
   +          4E       Suppress space before printing
   1          F1       Skip to channel  1
   2          F2       Skip to channel  2
   3          F3       Skip to channel  3
   4          F4       Skip to channel  4
   5          F5       Skip to channel  5
   6          F6       Skip to channel  6
   7          F7       Skip to channel  7
   8          F8       Skip to channel  8

   9          F9       Skip to channel  9
   A          C1       Skip to channel 10
   B          C2       Skip to channel 11
   C          C3       Skip to channel 12
```

Figure 22. Valid ASA Control Characters

Hex codes X'C1' and X'C3' are used in both machine code and ASA code. CMS recognizes these codes as ASA control characters, not as machine control characters.

Hex code X'5A' is recognized as only a machine code character. This code is used with a composed page data stream record.

Obtaining Device Information

Use the CMSDEV macro in conjunction with the PRINTL macro to obtain the device characteristics of the virtual printer. When the CMSDEV macro completes, the defined 12-byte storage area contains the device characteristics.

If the virtual device exists, the first four bytes contain:

Bytes
: **Virtual Device Information**

0
: Type class

1
: Type

2
: Status

3
: Flags

If the virtual device is associated with a local real device, bytes four through seven contain:

Bytes
: **Local Real Device Information**

4
: Type class

5
: Type

6
: Model number

7
: Current device line length for a virtual console, or the device feature code for other devices.

If the virtual device is associated with a remote real device, bytes four through seven contain:

Bytes
: **Remote Real Device Information**

4
: Type class

5
: Type for a remote 3270 console

6
: Model number for a remote 3270 console

7
: Current device line length for a remote virtual console.

If the virtual device is a local virtual console or a remote 3270 virtual console with an unknown address (*device* specified as CONS), bytes eight through eleven contain:

Byte
: **Information**

8
: The terminal code bits defining the type of virtual console and the translate table the console is using.

9
: Reserved

10-11
: Virtual device number

For virtual devices other than CONS, bytes eight through eleven contain:

Bytes
: **Information**

8
: Reserved

9
: Reserved

10-11
Virtual device number

For more information on device codes, see Appendix C in the *z/VM: CP Programming Services*.

Punching

Use the PUNCHC macroinstruction to write a line to the virtual punch at address X'000D'. Note that you must issue the CP CLOSE command to close the virtual punch file. Issue the CLOSE command either from your program (using CMSCALL) or from the CMS environment when your program completes execution. The punch is closed automatically when you log off or when you use the CMS PUNCH command.

The following example shows how to use (a) the PUNCHC macro to punch two lines and (b) the CMSCALL macro to invoke the CLOSE PUNCH command:

```
         PUNCHC  LINE1
         PUNCHC  LINE2
         CMSCALL PLIST=QPLIST
         .
         .
         .
         DS    0F
LINE1    DC    CL80'PUNCH THIS LINE FIRST'
LINE2    DC    CL80'PUNCH THIS LINE LAST'
QPLIST   DC    CL8'CLOSE'
         DC    CL8'000D'
         DC    2F'-1'
```

Note: No stacker selecting is allowed. The line length must be 80 characters.

Reading

Use the RDCARD macroinstruction to read a line from the virtual reader at address X'000C'.

Using the CMS Internal I/O Buffer

If you specify RDAHEAD=YES, CMS reads as many lines as it can into an internal buffer. As you issue RDCARD macros (starting with the first), CMS moves data from its internal buffer into the buffer area specified by your program. When the internal buffer is empty, CMS fills it up. This process continues as long as your program issues read requests or end-of-file is reached.

Note: If you process RDCARD with RDAHEAD=YES and the virtual card reader is closed before an error condition is detected (other than wrong-length record, RC=5), lines may still remain in the buffer. Subsequent RDCARD calls return the next available lines from the internal buffer until it is empty. Changes in the status of the virtual card reader are not recognized until the buffer is empty and the next physical read is performed. For most applications that read to end-of-file, RDAHEAD=YES should be specified.

To insure that the internal I/O buffer is released and that the next RDCARD request will read from the virtual reader, not the internal buffer, issue RDCARD with RDAHEAD=CANCEL and a length of zero.

RDAHEAD=NO is forced if the logical record length is greater than 2028, or if there is insufficient storage to allocate the internal I/O buffer.

Usage Notes

1. When the macro completes, register 0 contains the length of the card that was read.
2. No stacker selecting is allowed.
3. You may not use the RDCARD macro in jobs that run under the CMS batch machine.
4. If the reader file being processed contains carriage control characters, the RDCARD macro returns the records with the carriage control characters stripped off.

Tape Handling Macros

CMS provides several macros to help you use tape drives, including:

- TAPECTL — Positions the specified tape.
- TAPESL — Processes IBM standard HDR1 and EOF1 labels.
- RDTAPE — Reads a record from the specified tape drive.
- WRTAPE — Writes a record on the specified tape drive.
- TEOVEXIT — Sets up and clears a CMS end-of-volume tape exit.

For more information on these macros, see *z/VM: CMS Macros and Functions Reference*.

Tape Labels in CMS

CMS provides tape label processing for both ANSI and IBM standard labeled tapes in OS simulation. This support includes:

- For ANSI labels (AL) and ANSI user labels (AUL), translate input labels from ASCII to EBCDIC and translate output labels from EBCDIC to ASCII.
- Check IBM standard labels (SL) or ANSI labels on input tapes to ensure that the correct volume is mounted, as well as identify, describe, and protect the data being processed. For AL and AUL, input tapes are also checked to make sure that the labels are Version 3 ANSI[9] level.
- Check the existing SL, SUL, AL, or AUL, labels on output tapes to ensure that the correct volume is mounted and prevent overwriting of vital data.
- Create and write new SL, SUL, AL, or AUL labels on output tapes. For AL and AUL labels, only Version 3 ANSI[9] labels are written on output tapes.
- Specify exits for processing tapes with nonstandard labels during execution of CMS macro simulations and some CMS tape operation commands. CMS processes all tape labels.

Limitations

For CMS tape label processing:

- Processing of multivolume files on tapes is supported only under OS QSAM and BSAM simulation for ANSI and IBM standard labels. With the DISP MOD option, processing is only valid for the IBM standard labeled tape currently mounted.
- Label processing is not included for any of the TAPE command functions except for DVOL1 and WVOL1 which process VOL1 labels.

Initiating Label Processing

You must start all your own tape label processing. The commands you use to do this depends on the type of environment your program is running in.

- For programs running in the OS simulated environment:
  - You can specify that you have a labeled tape to process using FILEDEF.
  - You can define ANSI or IBM standard labels to be written or checked using LABELDEF.
- For programs running in the DOS simulated environment:
  - You can specify that you have a labeled tape to process using DOS DTFMT.
  - You can define IBM standard labels to be written or checked using LABELDEF. This command must be used to define the tape label you will be processing.
- For programs running in the CMS environment:

[9] Support for Version 3 ANSI refers to the code set defined by ANSI X3.27-1978, level 4.

- You can specify that you have an IBM standard labeled tape to process using the TAPESL macro.
- You can define ANSI and IBM standard labels using LABELDEF. This command must be used to define the tape label you will be processing.
- You can write or check ANSI and IBM standard labels using the TAPE command.

After label processing is requested, it automatically occurs and there is no interaction between you and CMS unless an error occurs. See "Error Processing" on page 170 for a discussion of error processing.

Label Processing in OS Simulation

The following table illustrates the basic tape layout for ANSI and IBM standard labels and a description of each label identifier.

VOL1	HDR1	HDR2	UHL1	UHL2	TM	DATA SET	TM	E0F1	E0F2	UTL1	UTL2	TM	TM

Label Identifier
Label Descriptor

VOL1
Volume label

HDR1 and HDR2
Data set header labels

EOV1 and EOV2
Data set trailer labels (end-of-volume - not shown)

EOF1 and EOF2
Data set trailer labels (end-of-file)

UHL1 through UHL8
User header labels (optional)

UTL1 through UTL8
User trailer labels (optional)

Figure 23. Basic Tape Layout for ANSI and IBM Standard Labels

For more information about the contents of these labels, see the *MVS/XA Magnetic Tape Labels and File Structure Administration*.

Types of Label Processing

If you are running an OS simulation program and using OPEN and CLOSE macros, specify the type of label processing you want in a FILEDEF command for a given file. Detailed information about the FILEDEF command is found in the *z/VM: CMS Commands and Utilities Reference*. The types of processing you can specify with FILEDEF TAPn command are:

SL
Indicates you are using IBM standard labels.

SUL
Indicates you are using IBM standard user labels.

AL
Indicates that you are using ANSI labels.

AUL
Indicates that you are using ANSI user labels.

NL
Indicates that your tape has no ANSI or IBM standard labels. (A file will not be opened if you specify this for a tape with a VOL1 label.)

BLP
: This tells CMS not to process the tape label, however, position the tape at the specified file before processing the data records in the file.

LABOFF
: This is the default and indicates that there is no CMS tape label processing for this tape file.

NSL
: Indicates you are using a nonstandard label process. You must have already written a routine to process nonstandard labels. The name of this routine must be specified with the NSL option on FILEDEF. An example of the nonstandard label processing is given in "Nonstandard Label (NSL) Processing" on page 160.

Examples of NL, BLP, and LABOFF processing are given in "Nonstandard Label (NSL) Processing" on page 160, "Bypass Label (BLP) Processing" on page 159, and "Label Off (LABOFF) Processing" on page 160.

IBM Standard Label Processing

Use the FILEDEF command to specify label processing for IBM standard labels (SL) or IBM standard user labels (SUL). For example,

```
FILEDEF PINTO TAP1 SL
```

defines PINTO as an IBM standard label tape file at the virtual address, 181 (specified with TAP1), to be positioned at the first data file (see number **1** in Figure 24 on page 153).

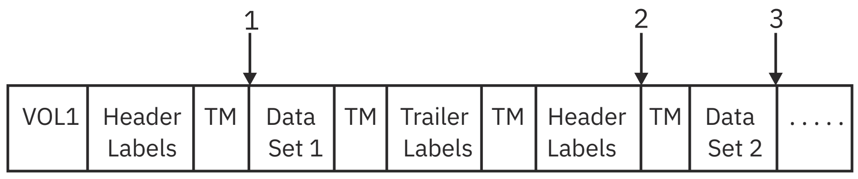


Figure 24. Basic Tape Layout

To specify an SUL tape, use the command,

```
FILEDEF PINTO TAP1 SUL
```

Note: Any option specified on the FILEDEF command has the same results for both SL and SUL tapes.

Writing VOL1 Tape Information

You can write VOL1 label information using the TAPE command. For example,

```
TAPE WVOL1 CATS (TAP1 SL
```

will write the volume serial number, CATS, to the tape located at virtual address, 181.

Displaying VOL1 Tape Information

Use the TAPE DVOL1 command to display information about a tape label. If you want to display the VOL1 information for the tape in the previous example, use the command

```
TAPE DVOL1 (TAP1 SL
```

and the following will be displayed:

```
VOL1CATS
```

Checking the Volume Serial Number

To make sure that the tape volume you requested is mounted, check the volume serial number (volid) using the FILEDEF or LABELDEF command. When the file is opened the volid specified is compared to the volume serial number in the VOL1 label. If they are the same, any read or write will be allowed. For example,

```
FILEDEF SIAMESE TAP1 SL VOLID CATS
```

When the file associated with the ddname, SIAMESE, is opened, the tape mounted at TAP1 will be checked for a volid of CATS.

Positioning Your Tape

To position your tape at data sets other than the first, use the *n* parameter on FILEDEF to specify the position of the file you want. For example, to position the tape at the second data set (number **2** in Figure 24 on page 153), use the command,

```
FILEDEF MUSTANG TAP1 SL 2
```

The DISP MOD option can also be used to position your tape. This option is only valid for the tape currently mounted; no volume switching is done. Following is an example of positioning a tape with DISP MOD.

```
FILEDEF PARROT A TAP1 SL 2 (DISP MOD
```

will position the tape at the end of the second file (number **3** in Figure 24 on page 153), ready to add new records.

You can also use the FILEDEF LEAVE command to position your tape.

Note: If you specify a file position number (*n*) that exceeds the number of files on the tape, the results are unpredictable.

Writing HDR1 Information on Output

HDR1 labels contain information about the data set immediately following it. To write information to the HDR1 label, use the LABELDEF command. For example,

```
FILEDEF SHEPARD TAP1 SL 3 VOLID DOGS
LABELDEF SHEPARD FID GERSHEP EXDTE 8906 SEC 1
```

When the file associated with the ddname, SHEPARD, is opened for output, the tape will be positioned to the third data set file. The values specified with LABELDEF (including the file identifier, expiration date, security level, and so on) will be written to the HDR1 label for the third data set file. See "LABELDEF Command" on page 169 for information on the default values written to the HDR1 label.

Checking HDR1 Information on Input

You can specify fields in HDR1 labels to be checked on input by using the LABELDEF command. This lets you check to see if you are positioned at the correct data file, have the correct security value, and so on. For example,

```
FILEDEF SIAMESE TAP1 SL 3 VOLID CATS
LABELDEF SIAMESE FID SIAM SEC 1
```

When the file associated with the ddname, SIAMESE, is opened for input, the tape volume at TAP1 is checked for the volid, CATS, and the tape is positioned to the third data set. The values specified on the LABELDEF command are checked for the third data set to make sure the file identifier is SIAM and its security access level is 1.

For more information on LABELDEF, see "LABELDEF Command" on page 169. For more information on security values, see the *MVS/XA Magnetic Tape Labels and File Structure Administration*.

Specifying Multiple Tapes

If you want to read an SL file that is contained on two tapes, you can use the FILEDEF command with the ALT option. ALT specifies an alternate tape drive to be used when an EOV condition occurs on the primary tape drive. (See "End-of-Volume and End-of-File Label Processing for OS" on page 157 for information on the EOV condition.) For example,

```
FILEDEF PIG TAP1 SL 3 VOLID ANIMALS (ALT TAP2
```

specifies that the file with a ddname of PIG is the third file on the tape located on TAP1. This is considered the primary tape drive. When an EOV condition is encountered on TAP1, the rest of the file, PIG, can be found on the tape located on TAP2.

You can also specify multiple tapes using the LABELDEF command. For example, if you issue the command,

```
LABELDEF POLLY VOLID ?
```

you receive the response:

```
DMSLBD441R Enter VOLID information
```

Now you can enter all the volids that must be used in order to process the file, POLLY. Following the last volid, enter a null line. If you enter the values:

```
BIRD1
BIRD2
null line
```

it will indicate that the file, POLLY is located on 2 tapes with the volids of BIRD1 and BIRD2. You can enter up to 8 volids on one line.

ANSI Label Processing

CMS can process both Version 3[10] ANSI labels and ANSI user labels. Except for the translation between ASCII and EBCDIC code, ANSI label processing is the same support provided for IBM standard labels and standard user labels. See "IBM Standard Label Processing" on page 153 for information in IBM standard label processing.

Use the FILEDEF command to specify label processing for ANSI labels or ANSI user labels. For example, use:

```
FILEDEF QUARTER TAP1 AL
```

to specify ANSI labeled tapes, and for ANSI user labeled tapes use:

```
FILEDEF QUARTER TAP1 AUL
```

ANSI tapes are recorded in 7-bit ASCII code (according to ANSI Version 3[10]). As part of the label processing, CMS translates data from ASCII to EBCDIC format when files are read from an ANSI tape. Files are translated from EBCDIC to ASCII format when written to an ANSI tape. This translation is supported for:

1. AL, AUL, LABOFF, BLP, and NL tape label processing
2. Record formats of F, FB, FS, FBS, D, DB, DS DBS.

For more information on ANSI label tape processing, see the *MVS/XA Magnetic Tape Labels and File Structure Administration*.

[10] Support for Version 3 ANSI refers to the code set defined by ANSI X3.27-1978, level 4.

Translating Between ASCII and EBCDIC Code

Use the OPTCD Q option of the FILEDEF command to indicate translation between ASCII and EBCDIC code. For example,

```
FILEDEF WELSH TAP3 NL (OPTCD Q
```

specifies that a nonlabeled tape on tape drive TAP3 will be processed. The OPTCD Q indicates that the information on the tape will be translated from ASCII to EBCDIC on input and from EBCDIC to ASCII on output. If a tape is specified as ANSI label or ANSI user label, then the OPTCD Q is automatically assumed.

For more information on reading and writing records, see "Reading and Writing CMS Files" on page 137.

Writing, Displaying, Checking ANSI Tape Labels

See "IBM Standard Label Processing" on page 153 for information on writing, displaying and checking tape labels. The support provided for IBM standard labels is the same for ANSI labels.

Protecting Data

Use the LABELDEF SEC command to read and write security values of a tape label. In addition to the 0, 1, and 3 security levels, ANSI labels may also contain security levels of A to Z (uppercase).

If you are reading a tape, CMS will compare the SEC value specified in the LABELDEF command to the SEC value written in the label; if the two values are equal, you will be allowed access. To write a security value to a tape, you could use the command:

```
LABELDEF VOLID HUNTER SEC A
```

This will write a security value of 'A' to the tape with a VOLID of HUNTER.

Note: To specify ANSI security values A-Z, or a VOLID with ANSI special characters, LABELDEF must be issued after the FILEDEF command. If a FILEDEF command has not been issued before the LABELDEF, the tape is assumed to be an IBM standard labeled tape.

Considerations for Standard Label Processing

When processing ANSI or IBM standard labeled tapes in OS simulation, the following applies:

- Multivolume tape processing does not occur if you enter the TEOVEXIT macro. TEOVEXIT is ignored for AL tapes.
- Multivolume tape processing only applies to CMS OS QSAM and BSAM simulation.
- During end-of-volume processing, the NOEOV operand of the FILEDEF command is ignored.
- The VOLSEQ operand of the LABELDEF command is ignored.
- Existing VOL1 labels are automatically rewritten for density incompatibility. However, VOL2 - VOLn and user header labels are not rewritten.

Considerations for User Label Processing

To process ANSI or IBM standard user labels in OS simulation, you must do the following:

1. Specify the file as AUL or SUL in a FILEDEF command.
2. Provide routines to process the user standard labels in your program.
3. Specify the address of the user label routines using the EXLST parameter in the DCB for the file. See the *MVS/XA Data Administration Guide* for information on how to establish a DCB EXLST and the exact linkage for communication between user label routines and the operating system. This exact linkage should be used under CMS with the following exceptions:
   a. There is no support for code X'06' EOV EXIT routine.

   b. For input labels, return codes 8 and 12 from the user routine are not supported. If an input return code is not 0, it is treated as if it were 4.

4. Note that your standard user label routines do not perform any I/O. They set up an output label for writing, but the CMS tape label processing routines actually write out the label. For input, the CMS label processing routines read in your user standard label but then give control to your routine to check the label.

Volume Label and Header Label Processing for OS

After you have set up your descriptive information for an ANSI or IBM standard labeled tape file, you can run a regular OS simulation program under CMS. Unless an error occurs, label processing will continue without any interaction with you. The following steps describe how tape volume labels (VOL1) and header labels are processed under OS simulation.

For Input Files on ANSI or IBM Standard Labeled Tapes:

1. If you have specified a VOLID parameter on your FILEDEF or LABELDEF command for a file, the VOL1 label on the tape is checked whenever the file on that tape is opened. This is to ensure that the correct volume has been mounted. If the volume ID is specified on both LABELDEF and FILEDEF, the more recent specification is used. If no volume ID is specified, the VOL1 label is not checked.
2. After checking the volid, the tape is positioned according to the positional parameters specified on the FILEDEF command. The positional parameters include *n*, DISP MOD, and LEAVE; the default is the first file on the tape.
3. HDR1 labels are checked at open time. This includes checking the file identifier (FID) or any other information specified in your LABELDEF command to make sure the correct file is accessed. If you do not explicitly specify a field for input, the field is not checked.
4. HDR2 labels are processed, if they exist, to determine certain data set characteristics of the file such as record format and block length.
5. If specified, user header labels (UHLn) are processed. You must provide a routine to do this.
6. For read backward processing, data management uses the data set trailer labels (EOF1 and EOF2) as header labels and vice versa. EOF1 labels are processed before EOF2 labels.

For Output Files on ANSI or IBM Standard Labeled Tapes:

1. If you have specified a VOLID parameter on your FILEDEF or LABELDEF command for a file, the VOL1 label on the tape is checked whenever the file on that tape is opened. This is to ensure that the correct volume has been mounted. If the volume ID is specified on both LABELDEF and FILEDEF, the more recent specification is used. If no volume ID is specified, the VOL1 label is not checked.
2. The tape is then positioned according to the positional parameters specified on the FILEDEF command. The positional parameters include *n*, DISP MOD, and LEAVE; the default is the first file on the tape.
3. The existing HDR1 and HDR2 labels are checked at open time and new header labels are written. If the HDR2 label is missing or has a different density then the VOL1 label is rewritten along with the HDR1 and HDR2 labels.
4. If specified, user header labels (UHLn) are written. You must provide a routine to do this.

End-of-Volume and End-of-File Label Processing for OS

An end-of-volume or end-of-file condition occurs when:

- A tape mark is read.
- The physical end of a tape is reached.
- The FEOV (forced end of volume) macro is issued.

  Following is a description of the end-of-volume (EOV1 and EOV2) and end-of-file (EOF1 and EOF2) label processing.

For Input Files on ANSI or IBM Standard Labeled Tapes:

1. For end-of-file condition, the EOF1 labels are processed. Among other things, this processing ensures that the file was read correctly. After EOF1 processing, the exit routine specified in the DCB gets control.
2. For end-of-volume condition, the EOV1 labels are processed for the current tape volume and tape volume switching occurs.
3. EOF2 and EOV2 labels are bypassed.
4. If specified, user trailer labels are processed. You must provide a routine to do this.

For Output Files on ANSI or IBM Standard Labeled Tapes:

1. For IBM standard and ANSI labeled tape, if you are at the end of the file, an EOF1 label, an EOF2 label, and a tape mark are written. The tape is rewound.
2. If you are at the end of a tape and you need another tape to finish writing, an EOV1 label, an EOV2 label, and a tape mark are written at the end of the first tape. The tape is rewound, unloaded, and a message is issued indicating that an EOV label was written. Tape volume switching then occurs.
3. If you specified nonstandard labels, instead of writing the EOV label, a tape mark is written. Then the nonstandard label routine specified on the FILEDEF command is given control.
4. For BLP or NL files, only the ending tape mark is written.
5. OS simulation programs that contain a BSAM CHECK macro cause an abend when end-of-tape is detected, with code 001 after an error message. A BSAM program that does not use a CHECK macro has no way of detecting the end-of-tape condition. Such a program continues to try to write on the tape after it is rewound and unloaded. The program enters a wait state rather than continue running to a normal or abnormal completion. Therefore, you should always include a BSAM CHECK macro after the WRITE macro if you expect your program to reach end-of-tape. OS simulation BSAM users are also responsible for completing processing on a new tape with the same or a new job after an end-of-tape condition is detected.

End-of-Tape Processing

End-of-tape (EOT) processing occurs if the record after a tape mark is an EOV label. For EOT processing, if the tape is SL or AL, volume switching occurs. If the next tape has already been mounted on an alternate drive, processing continues. (See "Specifying Multiple Tapes" on page 155 for information on specifying alternate tape drives.)

If an alternate drive has not been specified, the system notifies the operator to mount the next tape and you will get a notice that volume switching is about to begin. For example, if you are reading a file with the following attributes:

- A ddname of DUKE
- On a tape volume with a volid of DOG002
- The tape is located at virtual address 182

you will receive the following message:

```
Attempting to change tape volume for ddname DUKE
To cancel the tape volume switch, type CANCEL
```

You can stop the tape volume switch at anytime by typing CANCEL. This notifies the tape operator that you do not want the tape mounted, and terminates any further execution. Otherwise, the system waits for the tape operator to mount the requested tape volume and issues the following messages at five minute intervals until the volume is mounted:

```
Message sent to userid OPERATOR:
Mount tape volume DOG002 on virtual 182 without a write ring;
Request number 1
```

```
(five minute interval)
Message sent to userid OPERATOR:
Mount tape volume DOG002 on virtual 182 without a write ring;
Request number 2
(five minute interval)
Message sent to userid OPERATOR:
Mount tape volume DOG002 on virtual 182 without a write ring;
Request number 3
(five minute interval)
Wait time for tape volume switch has almost expired; to
continue waiting, type EXTEND
```

Note: Your system programmer can extend the wait time to longer than five minutes by using the TVSPARMS macro. Refer to "OS/MVS Tape Volume Switching" on page 397 for more information on the TVSPARMS macro.

At this point you can give the tape operator more time to mount the tape by typing 'EXTEND'. The operator receives the tape mount prompts once again, beginning with request number one. Otherwise, the following message is displayed at your terminal:

```
Message sent to userid OPERATOR:
Mount tape volume DOG002 on virtual 182 without a write ring;
Request number 4
(five minute interval; type EXTEND if you need more time)
Wait time for tape volume switch has expired; tape volume
switch for volume DOG002 on virtual 182 canceled
```

The allotted time for the tape volume switch is over and the read program is terminated. If you still want to process the data, you must begin again by reentering the initial FILEDEF and LABELDEF commands.

Note:

1. For SL tapes, if you enter the TEOVEXIT macro, multivolume tape processing will not occur. For a description of the TEOVEXIT macro, see the *z/VM: CMS Macros and Functions Reference*.
2. Your system support personnel can replace the OS simulation multivolume support with the DMSTVI interface routine or some other tape management system. If the multivolume support appears to be different on your system, see your system administrator.

Note: If you are an OS simulation user and the NOEOV option was specified on your FILEDEF command, it is ignored at the end-of-volume processing. However, the program causes an abend if you use QSAM or include a BSAM CHECK macro after your WRITE macro. Without a CHECK macro, a BSAM program runs the tape off the reel when EOT is sensed and NOEOV is specified.

No Label (NL) Processing

You should specify NL in the FILEDEF command when you expect a tape does not contain any IBM standard tape labels. CMS reads your tape at the time a file is opened and does not open the file if the tape contains a VOL1 label as its first record. If the tape does not contain a VOL1 label, a file is opened and the tape is positioned by using the position parameter (n). For example, if you specify

```
filedef fileq tap1 nl 2
```

FILEQ is not opened if the tape on TAP1 (181) has a VOL1 label. If the tape does not have a VOL1 label, FILEQ is opened and the tape is positioned at the second file. If you do not specify a position parameter, the tape is positioned at the first file, (that is, the load point).

Bypass Label (BLP) Processing

You should specify BLP in the FILEDEF command to bypass tape label processing. CMS does not check your tape for an IBM standard tape label. It uses the position parameter you specified to position the tape during open processing. If you do not specify a position parameter, the default is 1. For example

```
filedef fileabc tap1 blp 2
```

positions the tape at the second file when it opens FILEABC.

Because CMS does not know whether files on the tape are label files or data files, the tape is positioned at what is physically the second file, regardless of file content. Any label files on the tape are included in counting files. Therefore, the FILEDEF command in the last example would position the tape at data set 1. (See number **1** in Figure 25 on page 160.)

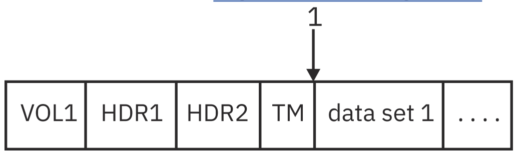


Figure 25. Positioning for BLP

For bypass processing, the header labels would be considered the first file and the data set the second file.

Label Off (LABOFF) Processing

You should specify LABOFF in the FILEDEF command if you want no positioning or label processing to occur during open processing. The position parameter (n) is not valid for LABOFF. If you specify LABOFF, and your tape is positioned at record 6 in the third file before you issue an OPEN macro, the tape is positioned at exactly the same record after open processing (record 6 in the third file); the tape is not moved. The following FILEDEF command does not move TAP2 (182) before processing the data in FILEB:

```
filedef fileb tap2 laboff
```

Nonstandard Label (NSL) Processing

To process nonstandard labels, you must write your own routine to read, write, and check the labels. If you have such a routine as a CMS TEXT or MODULE file, specify it after the NSL keyword. For example,

```
filedef DOGS TAP3 NSL CANINE
```

where CANINE is the name of a MODULE or TEXT file. If both a MODULE and a TEXT file exist with the file name specified in FILEDEF, the MODULE file is used.

The file name must be the name of the first CSECT in the program. It is to this point that control is transferred when the NSL routine gets control. If you do not have a TEXT or MODULE file with the NSL file name you specify, you get an error message.

Both OPEN and CLOSE load your routine if it is not already in storage and pass control to it at the time they are opening or closing the file. Your routine is then responsible for processing the tape labels. Your nonstandard label routine must do the actual reading and writing of tape labels as well as checking and setting up the label.

Note: This is one of several ways nonstandard label processing is different from standard user label processing. Because the CMS label processing routines do not know the size or format of your nonstandard labels, they cannot read or write the labels.

If you use a nonrelocatable MODULE file for an NSL routine, it is important that you create the MODULE file so that it starts at an address that will not let it overlay the program or command you are executing at the time the NSL routine is run. The reason for this restriction is that the nonrelocatable NSL routine is dynamically loaded while your program is running and your routine could overlay part of the program executing.

For the TAPEMAC and TAPPDS commands, starting the NSL routine at an address above X'21000' prevents such an overlay. If the NSL routine is run from your own program that is running in the user area, you must determine how big your program is and where the NSL MODULE file should be located to prevent overlay.

Note: You do not have to worry about overlaying other programs for NSL routines that are TEXT files. The CMS loader loads such files for you at an address that does not cause an overlay.

To ensure proper communication with the CMS system routines, you must use the linkage described in the following table when you write nonstandard label routines.

When an NSL tape label processing routine gets control, register 1 points to a 16-byte parameter list with the following format:

Table 21. Parameter List Layout

Disp	Len	Content
0	1	Type call
1	1	Caller ID
2	1	Tape MODESET
3	1	Reserved
4	4	TAPID
8	4	FCBSECT address
12	4	DCB address

The Type call field is a code telling the type of label processing being done:

```
x'00'     is OPEN input
x'04'     is OPEN output
x'08'     is CLOSE input
x'0C'     is CLOSE output
x'10'     is End Of Tape output
```

The Caller ID is a 1-byte code which is one of the following:

```
x'80'     Call by OS simulation
x'20'     Call by CMS TAPEMAC or TAPPDS commands
```

Tape MODESET byte communicates with the CMS tape I/O routines. It is a 1-byte hexadecimal code that depends on the type of tape (7, 9, or 18 track), tape density, and so forth. (You will probably pass this byte to the CMS tape controlling module to read and write your tape labels.) For more information on MODESET, see the TAPE command in the *z/VM: CMS Commands and Utilities Reference*.

DCB address is the address of the DCB for the tape file you are processing.

FCBSECT address is the address of the CMSCB (FCBSECT) for the tape file you are processing.

The following FCB fields might be of interest to an NSL tape user exit when the tape is opened or closed:

FCBDD
Data Definition Name as specified on the FILEDEF command.

FCBITEM
Item Count. This is a binary count of the number of data items read or written at this time.

FCBRFMT
Tape Device Recording Format. Below is a list of the complete TAPEIO recording format code values:

```
FDEFAULT EQU   X'00'          Any recording format at all
F9TRK    EQU   X'01'          Any 9 track recording format
FCOMP    EQU   X'02'          Any compacted recording format
FNOCOMP  EQU   X'03'          Any noncompacted recording format
FNRZI    EQU   B'11001011'    9T NRZI (9TRACK DEN(800))
FPE      EQU   B'11000011'    PE (9TRACK DEN(1600))
FGCR     EQU   B'11010011'    GCR (9TRACK DEN(6250))
F3480B   EQU   X'10'          3480 Basic (18TRACK DEN(38K)
F3480BB  EQU   X'DB'          Alternate 3480 fmt code, used by
                                old programs only
F3490B   EQU   X'20'          3490 Basic format
F3490C   EQU   X'30'          3490 Compacted format
F9346    EQU   X'50'          9346 format
F3480C   EQU   X'60'          3480 Compacted format
```

```
F3590B   EQU   X'40'          3590 Basic format
F3590C   EQU   X'70'          3590 Compacted format
```

FCBRECFM
Data Record Format as specified on the FILEDEF command.

FCBBLKSI
Tape Data Block Size as specified on the FILEDEF command.

FCBLRECL
Tape Data Logical Record Length as specified on the FILEDEF command.

FCBIOSW
Bit Flags Concerning I/O. The flags which are applicable to this exit are: FCBCLOSE, FCBCLEAV, FCBIOWR, and FCBIORD.

FCBIOSW2
Bit Flags Concerning I/O. The flags which are applicable to this exit are: FCBWRTSW and FCBTCLOS.

FCBTAPID
Tape Identifier. This field is TAPn as specified on the FILEDEF command.

FCBLABT
Tape Label Type. This field will always have bit FCBNSL set when the user gets control. FCBNSLMD may also be set if the user exit routine is a relocatable module.

FCBTPSW
Tape Bit Flags for Positioning as specified on the FILEDEF or FEOV macro.

FCBNSLNM
NSL Routine Name.

FCBLABPT
Address Pointer to LABSECT when a LABELDEF command is issued. For more information on LABSECT, see the *z/VM: CMS Macros and Functions Reference*. For more information on the LABELDEF command, see the *z/VM: CMS Commands and Utilities Reference*.

FCBBLKCT
Block Count for Tape File. This is a binary count of the number of unique blocks of data record items existing on the tape. This number is stored in the trailer labels after the data and corresponds to the total FCBITEM count.

For more information about the FCB, see the CMSCB MACRO in the *z/VM: CMS Macros and Functions Reference*.

Note: For the TAPEMAC and TAPPDS commands, the same interface is used, except that instead of the FCBSECT and DCB address fields, the eight character identifier specified in the ID=*identifier* field in the command is passed. This identifier lets you identify which file you are processing because the TAPEMAC and TAPPDS commands do not work with CMSCBs or DCBs.

Control is passed to your NSL routine by a BALR 14,15 instruction so register 15 contains the address of your routine when you receive control. Register 14 contains the address you should return to when you are finished processing the nonstandard labels. You can return with a BR 14 instruction. When you receive control, register 13 points to a save area in which to store the callers register. The save area linkage is standard OS/VS linkage. You receive control with a PSW key of X'E' that lets you modify only user storage. When you are finished processing, place a code in register 15 to the CMS label processing routine that called your routine. Place the value 0 (zero) in register 15 if there have been no errors and you want processing to continue normally and the data set to be opened. If you return a nonzero value in register 15, a message is issued to your terminal and the data set is not opened.

If you write the following FILEDEF statement

```
filedef tapfl tapl nsl readlab
```

and have a program called READLAB as a MODULE or TEXT file, your program receives control when the data set called *tapfl* is opened. When your program gets control, register 1 contains the address of the parameter list that is described in the preceding example. Using the data in this parameter list, you can read or write your own tape header labels. When the same data set is closed, your program again receives control and you can read or write your own trailer labels. Your program can test whether it is getting control for OPEN or CLOSE by examining the type call byte in the parameter list passed to you. If the type call byte is X'10', your NSL routine is run while you are writing an output data set and you have reached the reflective mark that indicates end of tape. You may wish to do special processing in this case. See "End-of-Tape Processing" on page 158 for more information on end-of-tape processing.

Tape Label Processing Differences

There are a some differences in the way CMS OS simulation processes tapes and the way OS/MVS processes them:

- If you are using OS/MVS and you do not specify any label parameter on your JCL statement, the default is SL or standard labels. When you use OS simulation under CMS and do not specify any label information on a FILEDEF statement, the default is LABOFF. LABOFF turns off label processing and nothing is done to position the tape or process labels. Thus, if you specify no label information on FILEDEF, the system processes your tape files exactly the same way they are processed on a CMS system that has no tape label processing facilities.
- You must specify CLOSE to process all trailer labels. No automatic CLOSE occurs at end of data or after reading a tape mark. There is no EOV monitor to process labels before a data set is closed. If an input tape is positioned at an EOF1 or EOV1 record when CLOSE is entered, the label is processed. If a tape file is closed before all data records are read, the trailer label is not processed. EOF labels are written onto tapes only at close time.
- There is no deferred label processing under OS simulation in CMS.
- When you have not specified a block count routine in your DCB EXIT list under OS/VS, the program abends when a block count error occurs. Under CMS, this condition produces a message that asks whether to abend the operation.
- Certain fields in HDR1 and EOF1 labels default to values different from those under OS/VS. These values can always be specified in a LABELDEF command if you do not like the default values. For example, the default for data set name in an output label under OS simulation is DDNAME and not DSNAME. The default data set sequence number is always one even when the data set is not the first data set on the tape. The default volume sequence number is always one. Read "LABELDEF Command" on page 169 to learn what the default values are under CMS. You can find what default values are in OS/MVS by reading the *MVS/XA Magnetic Tape Labels and File Structure Administration.*

  Note: You can always get exactly what you want written on a tape label by explicitly specifying the field on a LABELDEF command. For example, you can specify DSNAME as FID on such a command and have it written in the label instead of DDNAME.
- Default volume IDs (when you do not specify a volume ID in a LABELDEF or FILEDEF statement) in output HDR1 and EOF1 records under CMS will be CMS001 and will not be the actual volume serial in the VOL1 record already on the tape, unless you are processing SL tapes, in which case it will be the actual volume serial already on the tape. You should always specify the volume ID in FILEDEF or LABELDEF to be sure the information written is correct.
- Expiration date specification is always done in absolute form rather than by retention period. You can use either the 5-digit form (*yyddd*) or the 6-digit form (*cyyddd*), where *yy* is the year (00-99) and *ddd* is the day (001-366). In the 6-digit form, *c* is the century (9=1900s, 0=2000s, 1=2100s). CMS does not handle expiration dates specified by retention periods.
- When CMS reads a HDR1 label and finds an unexpired file, it always issues a message letting you enter ERROR or IGNORE. ERROR prevents opening the file in OS simulation. When the DISP MOD option of the FILEDEF command is specified for SL tapes, IGNORE lets you have the tape positioned at the end of the file, ready to add new records. Otherwise, IGNORE causes the existing record to be overwritten.
- The NSL routine linkage is quite different under CMS than in OS. (See "Nonstandard Label (NSL) Processing" on page 160 for more information.)
- Volume serial number verification occurs every time a file on a tape is opened under OS simulation unless the FILEDEF LEAVE option is used for multifile tapes.

- Existing VOL1 labels are automatically rewritten for density incompatibility in CMS as they are in OS/VS. However, VOL2 - VOLn and user header labels are not rewritten.
- The information from the HDR2 label is checked and merged with the FCB information and then with the DCB information. The merged information does not overlay existing data. Only the empty fields are filled with the information from the HDR2 label or the FCB.
- To maintain OS compatibility in the EOV2/EOF2 label, you must specify LRECL in the output FILEDEF.
- Multivolume tape processing only applies to OS QSAM and BSAM simulation for SL and AL tapes.
- Blank tapes used for output in CMS cause the tape to run off the reel if you define the tape file as SL, AL, or NL. The tape label processing routines try to read an existing VOL1 or HDR1 label before writing on the tape. Therefore, you should always use the TAPE command to write at least one tape mark (for NL tapes) or a VOL1 label (for SL, SUL, AL, or AUL tapes) before using the tape to write an output data set.
- If you specify a position parameter that is too big (that is, there are not that many files on the tape), the tape runs off the reel in CMS.

  For SL and AL tapes, positioning will occur across volumes. Positioning always uses the tape that is mounted at open time as the first volume of a multivolume group. File sequence values in labels are not used in positioning; positioning is based on the physical placement of the files on the tape.
- There are no user exits for user standard labels for EOV label processing in CMS.
- CMS does not support user return codes of 8 and 12 for input standard user labels. If the return code from a user routine is not zero after input label processing, CMS treats it as if the return code was 4. (See the *MVS/XA Data Administration Guide* for more information.)
- No count is kept of user standard labels read or bypassed in CMS. If more than eight such labels exist, the fact is not detected.
- User label processing routines do not receive control under CMS when an abend or a permanent I/O error occurs.
- If a CMS output tape is not positioned at a HDR1 label or a tape mark when label processing begins, error message 422 is issued. Under OS/MVS such conditions cause an abend.
- TCLOSE with the REREAD option causes a tape to be rewound under CMS and then forward spaced one file if the tape has standard labels. Under OS/VS, the tape is backspaced four files and forward spaced one file. REREAD for unlabeled tapes in CMS always causes a rewind.

For details on end-of-tape processing under CMS, see "End-of-Tape Processing" on page 158.

Label Processing in CMS/DOS

You specify the type of label processing you want in CMS/DOS on a DTFMT macro in exactly the same way you specify it when you want to run your program under VSE. See the "DTFMT Macro — Defines the File for a Magnetic Tape" on page 437 for details on CMS support for the DTFMT macro. beled tapes are only supported if you use the DTFMT macro. There is no support for labeled tapes in CMS/DOS for any other type. If you try to read labeled tapes with a DTFCP or DTFDI macro, input standard IBM header labels are skipped, but no other input labels are processed. Output tapes with standard labels have these labels overwritten with a tape mark. All tape work files are treated as output unlabeled files in CMS/DOS although they are defined by a DTFMT. Tapes used for such files have a tape mark written as the first record when the file is opened.

Unlabeled and Nonstandard Labeled Tapes

You define an unlabeled tape with the DTFMT parameter FILABL=NO. The tape file is processed as having no labels.

You define a nonstandard labeled tape with the DTFMT parameter FILABL=NSTD. You must also provide a routine to process your nonstandard labels in the LABADDR=parameter of the DTFMT. Tape processing in CMS for these files is the same as it is under VSE.

Standard Labeled Tapes

You define a standard label tape with the DTFMT parameter FILABL=STD. You also must supply a LABELDEF command to specify label description information. This command replaces the VSE TLBL card and is required for standard label processing under CMS/DOS. See "LABELDEF Command" on page 169.

To connect the LABELDEF command for a file with the DTFMT for the same file, you must use the same name to label your DTFMT as you use for a *filename* in your LABELDEF command. If you code a DTFMT macro in your program as

```
MT1 DTFMT     ...FILABL=STD
```

you must then supply the following type of LABELDEF command:

```
labeldef mt1 fid yourfile fseq...
```

You can put any description parameters you want on your LABELDEF command but the *filename* for it must be MT1 if you coded MT1 as the label on the DTFMT.

After you have set up your DTFMT and LABELDEF, you execute your CMS/DOS program. HDR1 labels are checked or written when an OPEN macro is run. A VOL1 label volume serial number is checked only if the tape is positioned at load point when the label processing begins and if you have specified a VOLID parameter on a LABELDEF statement for the file. Note, if NOREWIND is not specified in the DTFMT macro for the file, the tape is rewound so it is positioned at load point for label processing.

If you want to process user standard labels as well as standard labels in CMS/DOS, you specify FILABL=STD and also supply a LABADDR parameter in the DTFMT for the file. Control is then transferred to your label processing routines after standard labels are processed. The linkage to user standard label routines is exactly the same as in VSE.

End-of-Volume and End-of-Tape Processing in CMS/DOS

The following steps are taken when the end of volume is reached on a tape processed in CMS/DOS simulation.

For Input Files

- CLOSE processing checks EOF1 labels.
- If CLOSE or TAPESL processing reads an EOV1 label when it is expecting an EOF1 label, it issues a message. The EOV1 label is then processed exactly as if it were an EOF1 label. You must request that the operator mount a new tape and reopen a file if you want to continue processing the data.

For Output Files

1. If you specify that you have an IBM standard labeled tape file, a single tape mark is written to end your data. This occurs when end-of-tape is sensed on output while you are using regular access method macros to write the file. The tape mark is written immediately after the record that caused the EOT to be sensed.
2. Following this tape mark, CMS writes an EOV1 label and a single tape mark. It then rewinds and unloads your tape. A message is issued indicating that an EOV1 label was written.
3. CMS/DOS jobs are always canceled after an EOT condition is detected on output. To continue processing the tape, you must have a new tape mounted, run the same job over again or run a new job and reopen the file.
4. If you are a CMS/DOS user you always get the automatic output end-of-tape processing described above.

Tape Label Processing Differences

There are some differences in the way tapes are processed by CMS/DOS and the way they are processed by VSE:

- The tape error messages are CMS error messages and not VSE error messages. In some cases VSE lets the system operator reply NEWTAP to an error message. The system then waits for the operator to mount a new tape and continues processing with this new tape. Such a reply is never possible under CMS/DOS. In CMS/DOS, you usually can reply IGNORE to ignore a tape label error condition or CANCEL to cancel a job. NEWTAP is never allowed. In a few cases, CMS/DOS allows an IGNORE reply where VSE does not.
- You must specify CLOSE to process all trailer labels. No automatic CLOSE occurs at end of data or after reading a tape mark. If an input tape is positioned at an EOF1 or EOV1 record when CLOSE is run, the label is processed. If a tape file is closed before all data records are read, the trailer label is not processed. Output tapes have EOF records written only at CLOSE time. For nonstandard labeled tapes, your own routines do not receive control on input when a tape mark is read. You must enter a CLOSE macro in your EOFADDR routine to have the trailer labels processed.
- Certain fields in HDR1 and EOF1 labels default to values different from those in VSE. For example, the default volume serial number written in a HDR1 label is CMS001 and not the actual volume ID (volid) in the VOL1 label already on the tape. The default file sequence and volume sequence numbers are always one even when the file is not the first file on the tape. You should read "LABELDEF Command" on page 169 to learn what the default values are in CMS/DOS. You also can read *VSE/AF Tape Labels* to find what the default values are for VSE. If you do not like the default values, you can always specify the exact values you want in label fields in a LABELDEF command.
- Expiration date specification is always done in absolute form rather than by retention period. You can use either the 5-digit form (*yyddd*) or the 6-digit form (*cyyddd*), where *yy* is the year (00-99) and *ddd* is the day (001-366). In the 6-digit form, *c* is the century (9=1900s, 0=2000s, 1=2100s). CMS does not handle expiration dates specified by retention periods.
- VOL1 labels written in the wrong density are not automatically rewritten by CMS/DOS as they are by VSE.
- Blank tapes should not be used for tape files specified as FILABL=STD in CMS/DOS; they will run off the reel. Use the TAPE command to write a VOL1 label or a tape mark on a blank tape before using it for a STD file.
- Not all tape movement and label checking that occurs in VSE occurs under CMS. For example, when opening an output file, a VSE system expects the tape to be positioned at a HDR1 label or a tape mark. It then backspaces the tape to read the last EOF1 label on the tape. If it does not find the label it expects, it displays an error message. This check is not performed by CMS/DOS. If the tape is not positioned at a HDR1 label or a tape mark when output open processing begins, error message 422 is issued.
- After an EOV1 label is written (see "End-of-Tape Processing" on page 158), the tape is always rewound and unloaded under CMS/DOS. VSE lets a DTFMT parameter control if the tape is rewound.
- User label processing routines do not receive control when an I/O error occurs under CMS/DOS.
- Control is not passed to user standard label routines in CMS/DOS when EOT has been sensed on output and an EOV1 label has been written by the system routines.
- Work tapes are not checked for an expiration date when they contain standard labels under CMS/DOS. If a tape is to be opened as a work tape, CMS/DOS tests to see if it contains a VOL1 label. If it does, a dummy HDR1 label and a tape mark are immediately written on the tape after the VOL1 label. If the tape does not contain a VOL1 label, a tape mark is written at the beginning of the tape. VSE checks expiration dates on previously labeled tapes used as work tapes and gives the operator a chance to reject the tapes if the expiration date has not expired.

For more information on VSE and CMS/DOS tape label processing, see the *VSE/AF Tape Labels* and *VSE/AF Macro User's Guide*.

Label Processing Using CMS Macros and Commands

Several CMS macros and commands can be used for label processing in the CMS environment. These include, the TAPESL macro and the TAPEMAC, TAPPDS, TAPE, MOVEFILE, and LABELDEF commands.

TAPESL Macro

Use the TAPESL macro to process IBM standard HDR1 and EOF1 labels without using DOS or OS OPEN and CLOSE macros. You will probably use TAPESL with the RDTAPE, WRTAPE, and TAPECTL macros.

TAPESL processes only HDR1 and EOF1 labels. It does not perform any functions of opening a tape file other than label checking or writing. It is used both to check and to write tape labels. A LABELDEF command must be entered before running the program that contains this macro. The LABID parameter of the TAPESL macro specifies the name of the LABELDEF to be used. For example, if you use the macro

```
TAPESL HOUT,181,LABID=GOODLAB
```

in your assembler language program, you must supply a LABELDEF command for GOODLAB:

```
labeldef goodlab fid file10 fseq 4 exdte 002235
```

The tape must be correctly positioned (at the label to be checked or at the place where the label is to be written), before you run the macro. The TAPECTL macro can be used to position the tape. TAPESL reads or writes only one tape record unless you specify SPACE=YES for input. Then it spaces the tape to beyond the tape mark that ends the label file. TAPESL reads and checks a tape VOL1 label provided the tape is positioned at load point and you have specified a volume ID in your LABELDEF command.

TAPEMAC and TAPPDS Commands

TAPEMAC and TAPPDS have operands where you can indicate the type of label processing you want. The tape must be properly positioned (at the data file or label file you want) before you enter the command. The TAPE command can be used for positioning. A separate LABELDEF command is required for these commands if IBM standard label checking is desired. If SL label type is specified without a *labeldefid*, standard header labels are displayed on the terminal but not checked by the CMS label processing routines. The command

```
tapemac macfile SL (tap2
```

displays any standard labels that exist on your terminal while the series of commands:

```
labeldef maclab fid macro volseq 2 crdte 998102
tapemac macfile sl maclab (tap2
```

runs the CMS tape label processing routines. These routines check to see that your tape has a HDR1 label that has a file identifier of macro, a volume sequence number 2, and a creation date of 998102 (day 102 of year 1998). VOL1 labels are not checked during label processing by TAPEMAC and TAPPDS unless the tape is positioned at load point and you have specified a volume ID on your LABELDEF command. The DVOL1 function of the TAPE command can be used for volume verification before positioning the tape if the user does not want to start at the first file. These commands process only HDR1 labels; they skip HDR2, UHL, and all trailer labels without processing them.

To process nonstandard tape labels with TAPEMAC and TAPPDS, you use the same interface described in “Nonstandard Label (NSL) Processing” on page 160. The only difference is that instead of putting the CMSCB and DCB addresses in the parameter list, the ID parameter you placed in the command line is passed to your NSL routine.

```
tappds pdsfile cmsut1 * nsl superck id XYZ12345
```

passes the EBCDIC identifier XYZ12345 to your nonstandard label checking routine called SUPERCK. This identifier may be up to eight characters long and is left justified in bytes 8 to 15 of the parameter list. You can use the identifier to inform your NSL routine of what file you are processing.

TAPE Command

Use the DVOL1 function of the TAPE command to display the VOL1 label of a tape on your terminal. You can use this command to ensure the system operator has mounted the correct tape before you begin processing the tape. If the tape does not have a VOL1 label and you enter the TAPE command, you are informed that the VOL1 label is missing.

Do not use TAPE DVOL1 if you have a blank tape. If TAPE DVOL1 is entered and a blank tape is used, CMS will search the entire tape to find the label record; because the tape is void of any records, the tape will run off the end of the reel in a reel tape, or you will get an error message with a cartridge tape.

Use the WVOL1 function on the TAPE command to write a VOL1 label on a tape. You can specify a one-to-six character volume ID (volid) through this command and also a one-to-eight character owner field.

Examples of DVOL1 and WVOL1

You can display or write VOL1 labels for ANSI and IBM standard labeled tapes. If you want to display the VOL1 label from an IBM standard labeled tape, you can use the following command:

```
TAPE DVOL1 (TAP3 SL
```

This will display the VOL1 label of the tape located at the virtual address, 183.

If you want to write a VOL1 label for an ANSI labeled tape, you could use the command:

```
TAPE WVOL1 PERSIAN (TAP3 AL
```

This would write the volume serial number, PERSIAN, to the ANSI labeled tape located at virtual address, 183.

MOVEFILE Command

Moving Labeled Tape Files

You can use the MOVEFILE command to move labeled tape files if these files are defined as labeled by the FILEDEF command. The MOVEFILE command supports only SL, AL, NSL, BLP, NL, and LABOFF processing. SUL files are processed as SL files and AUL files are processed as AL files; and no user exits are taken.

You can also use the MOVEFILE command to display tape labels on your terminal if you want to see what these labels look like. The following sequence displays the VOL1 and first HDR1 labels on TAP4 if the tape has standard labels:

```
filedef in tap4
filedef out term
tape rew (tap4
move in out
```

Copying Sequential Files into CMS Files

The MOVEFILE command can also copy sequential tape files into CMS files (minidisk files or files that reside in an SFS directory), or sequential CMS files onto tape. It can be particularly useful when you need to copy a file from a tape and you do not know the format of the tape.

To use the MOVEFILE command, you must first define the input and output files using the FILEDEF command. For example, to copy a file from a tape attached to your virtual machine at virtual address 181 to a CMS minidisk, you would enter:

```
filedef input tap1
filedef output disk tape file a
movefile input output
```

This sequence of commands creates a file named TAPE FILE A1. Then use CMS commands to manipulate and examine the contents of the file.

LABELDEF Command

LABELDEF can be used to write or check label information on both ANSI and IBM standard label tapes. The LABELDEF command specifies the exact data you want written in certain fields of a HDR1 or EOF1 tape label for output. If you do not explicitly specify a field for output, a default value is used.

LABELDEF can also be used to specify fields in the same labels that you want checked on input. Even if you do not specify a value for the file identifier field, it is checked to make sure it is not zeros or blanks.

The following LABEDEF command could be used for either an input or output file.

```
labeldef abc fid master volseq 1 exdte 005364
```

If used for output, MASTER is written to the HDR1 label in the file identifier field, 1 is written in the volume sequence number field and 005364 (day 364 of year 2005) is written in the expiration date field. Default values are written in the HDR1 fields that are not specified.

If the same command is used for input, CMS checks the file identifier, volume sequence number, and expiration date in an input HDR1 label. No other fields in the label are checked.

Default values for HDR1 labels are as follows:

FID
For OS simulation, the ddname is specified in the FILEDEF command for the file.

For CMS/DOS, FID is the DTFMT symbolic name.

For TAPESL macro, FID is the *labeldefid* specified in the LABID parameter.

VOLID
is CMS001.

For OS simulation, the actual volid from the tape mounted is used if processing an SL or AL tape file.

In multi-volume processing, for subsequent volumes, the actual volid from the tape mounted is used in the *volume serial number* field in the volume label (VOL1), and the volid of the first volume is written to the *data set serial number* field in the HDR1 label.

VOLSEQ
is 0001.

FSEQ
is 0001.

GENN
is blanks.

GENV
is blanks.

CRDTE
is the date when the label is written.

EXDTE
is the date when the label is written.

SEC
is 0 for IBM standard labeled tapes and blank for ANSI labeled tapes. (If a 0 is specified for an ANSI labeled tape, the SEC code will be set to blank, however, a blank cannot be specified with the SEC option.)

The *filename* on the LABELDEF command connects your label definition to a file defined elsewhere. This is why you specify different data for *filename* depending on the type of tape label processing you are doing. *Filename* is ddname for OS simulation, DTFMT symbolic name for CMS/DOS and labeldefid for TAPESL.

The LABELDEF command takes the place of the VSE TLBL statement for CMS/DOS.

Error Processing

When the standard label processing routines find errors or discrepancies on tape labels, they send a message to the CMS terminal user who is processing the tape. After an error message is issued, you can ask the system operator to mount a new tape, use the TAPE command to position the tape at a different file, or specify your label description information again. If you are a terminal user and want another tape mounted, you send the system operator a message telling him or her what tape to mount.

Some errors cause program termination and others do not. The effect of tape label processing errors depends on both the type of error and the type of program (that is, CMS/DOS, OS simulation, CMS command, and so forth) that runs the label processing. The following are general guidelines on error handling:

- Messages identifying the error are always issued.
- Under OS simulation, tape label errors result in open errors. These errors prevent a tape file from being opened. They do not necessarily end a job. Errors in trailer labels (except block count errors) have no effect on processing.
- In CMS/DOS, the terminal user is generally given two choices: ignore the error or cancel the job. The new-tape option is not allowed.
- The CMS commands TAPEMAC and TAPPDS terminate with a nonzero return code after a tape label error.
- Certain error situations such as unexpired files and block count errors for OS simulation let you ignore the error and do not cause open errors. In these cases, you enter your decision at the terminal after you are notified of the error.
- Errors that occur during the loading of an NSL routine cause an abend (code 155 or 15A). A block count abend gives an error code of 500.

In all cases, after an error has been detected and diagnosed, you must decide what to do. You may wish to have a new tape mounted and then reenter the command or you may want to respecify your LABELDEF description if it was incorrect. You can also use the TAPE command to space the tape to a new file if it was incorrectly positioned.

Using Tape Library Dataservers under CMS OS Simulation

Tape Library Dataserver machines (such as the 3494, 3495, and 3595) use a robotic tape operator to automatically select, mount, and demount tapes in a mechanically controlled tape library. This library and its attached tape cartridge drives (such as the 3480, 3490, and 3590) are under the control of the Removable Media Services (RMS) system. z/VM provides an interface to the RMS system through DFSMS/VM.

If tapes are to be used under CMS OS simulation on a Tape Library Dataserver, z/VM requires that:

- DFSMS/VM and RMS must be installed.
- The RMS CSL library (FSMPPSI CSLLIB) must be accessed. In addition, the following command must be issued to make the RMS CSL routines available to CMS:

  ```
  rtnload * (from fsmppsi
  ```

- The RMS system administrator must select one of the 16 SCRATCH*x* (where *x* is 0-F) library categories as the default SCRATCH processing category pool.
- The user should make sure that the default SCRATCH pool of tapes has enough physical tapes currently assigned to it to meet any application program needs.
- If unique VOLIDs are specified on a LABELDEF or FILEDEF statement, these VOLIDs must reside within the tape library to enable the Dataserver to mount the tapes.
- If the user premounts a tape on a Dataserver device with the DFSMS/VM MOUNT command, and the tape is to be used for any type of output, the Target Category should be set to VOLspecific in the MOUNT

command parameters. This corresponds to the automatic system Target Category setting for output tape mounts.

CMS provides the following interfaces:

- If the LIBSRV option is specified on the FILEDEF command to indicate that tape mounts should be done on a Tape Library Dataserver machine, OS simulation calls the RMS interface to mount the tapes automatically for the user. It is suggested that this option be used whenever a Tape Library Dataserver is being used, to allow the system to mount all the tapes automatically for the user.
- If the tape drive currently in use is found to be under the control of a Tape Library Dataserver, or the FILEDEF command has been issued with the LIBSRV option, OS simulation attempts to get subsequent multivolume tapes mounted automatically for the user through the native DMSTVS mounting service and the CMS native rewind and unload tape processing functions by calling the RMS FSMRMDMT (Demount) and FSMRMMNT (Mount) CSL routines.
- RUN (rewind and unload) function processing for the CMS TAPE or VMFPLC2 command or the TAPECTL macro calls the RMS FSMRMDMT (Demount) CSL routine if a Tape Library Dataserver is found to be controlling the tape drive that the RUN was issued against.
- Input tapes remain in the same category on demount, but output tapes are moved to the VOLspecific category to prevent accidental tape overwrite on the next scratch tape mount from the default SCRATCH tape pool.

Chapter 12. Interrupt Handling

CMS interrupt handling routines can be used in an XC, XA, or 370 virtual machine. These routines make it easier for application programs to handle interrupts (or easier for programmers to code interrupt handlers). This chapter discusses:

- Manipulating the PSW Interrupt Mask
- External interrupt handling
- I/O interrupt handling
- SVC interrupt handling
- Machine-check interrupt handling
- Program interrupt handling.

Note: Only CMS levels prior to CMS Level 12 can execute in a 370 virtual machine.

Manipulating the PSW Interrupt Mask

Because of PSW format differences between System/370 architecture and ESA/XC and 370-XA architecture, the SSM (SET SYSTEM MASK) assembler instruction produces undesirable results in an XC or XA virtual machine. The ENABLE macro provides an architecture-independent way to set the interrupt mask in the PSW. The macro expansion includes logic to (a) check for the type of virtual machine, (b) build the appropriate interrupt mask, and (c) issue instructions appropriate to the architecture. If you specify MODE=NO370 on the ENABLE macro, then those instructions needed only for the 370 virtual machine are **not** generated.

Converting from the SSM Instruction to ENABLE

To make migration easier and to exploit the ENABLE macro's dual-path code, replace SSM instructions with the appropriate ENABLE macro. The following table suggests replacements.

Table 22. Replacing SSM Instructions with ENABLE Macros

SSM	ENABLE
=X'00'	INTTYPE=NONE
=X'FF'	INTTYPE=ALL
=X'01'	INTTYPE=EXTERNAL
=X'FE'	INTTYPE=IO
=X'80'	INTTYPE=CONSOLE
=X'7E'	INTTYPE=NONCONIO
=X'81'	INTTYPE=(CONSOLE,EXTERNAL)
=X'7F'	INTTYPE=(NONCONIO,EXTERNAL)

Examples

Example 1

To disable I/O and external interrupts, code the ENABLE macro as follows:

```
ENABLE INTTYPE=NONE
```

Example 2

To enable all I/O and external interrupts, code the ENABLE macro as follows:

```
ENABLE INTTYPE=ALL
```

Example 3

To generate reentrant code, use a combination of MF parameters such as

```
ENABLE INTTYPE=ALL,MF=(L,(reg))
```

to identify the mask settings, and

```
ENABLE MF=(E,(reg))
```

to execute the instructions to set the interrupt masks. The specified register should contain the address of a two fullword work area. Such a work area should not be within the program that is to be reentrant. It might be in a save area, or it could be obtained with the CMSSTOR macro.

Note: Issuing the CP SET 370ACCOM ON command will allow SSM to produce results consistent with a 370 virtual machine. See *z/VM: CP Programming Services* for more information on how to run your 370-only CMS applications in an XA or XC virtual machine. See *z/VM: CP Commands and Utilities Reference* for information on the CP SET 370ACCOM command.

External Interrupt Handling

CMS external interrupt facilities provide:

- Complete interrupt status information in a common format.
- Notification of the occurrence of the interrupt according to the programmer's need (notification can be immediate or synchronized using event control blocks (ECBs)).
- A default external interrupt handler that allows you to abend a program or continue processing.
- A macro interface (the HNDEXT macro) that allows you to define external interrupt handling routines. You can use HNDEXT to define handlers for specific interrupt codes or to define your own default external interrupt handler to process external interrupts that do not have a specific handler.

  In addition, the HNDEXT macro provides parameters that allow you to specify whether the interrupt handler is to be cleared at end-of-command (the KEEP parameter), specify whether the handler survives abend processing (the SYSTEM parameter), specify an error return address (the ERROR parameter) and specify various macro formats (the MF parameter).

  Note: If you specify SYSTEM=YES and KEEP=YES for an interrupt handler, you should place the entry address of the routine (rtnaddr) in a nucleus extension or unpredictable results will occur.

External Interrupt Handling Overview

External interrupts provide a means for your virtual machine to respond to various signals originating from either inside or outside its configuration. For example, all of the following architected sources can generate external interrupts:

- Interrupt key
- Malfunction alert
- Emergency signal
- External call
- TOD-clock sync check
- Clock comparator
- CPU timer

- Service signal.

Note that many of these interrupts do not apply to CMS. For example, emergency signals, malfunction alerts, service signals, and external calls apply to multiprocessing, which CMS does not support.

Other External Interrupt Sources

In addition to the architected sources, external interrupts are generated by APPC/VM, IUCV, VMCF, the CP EXTERNAL command, and by CP for events related to data space support (interrupt codeX'2603'). The X'2603' interrupt code can indicate either a page-fault notification or the completion of an asynchronous save operation. The X'2004' interrupt code notifies CMS of a time zone change that has been initiated by a privileged user entering the CP SET TIMEZONE command. For information about the CP SET TIMEZONE command, see *z/VM: CP Commands and Utilities Reference*.

Enabling and Disabling External Interrupts

To enable or disable your virtual machine for all external interrupts, you can use the ENABLE macro (see "Manipulating the PSW Interrupt Mask" on page 173). To enable or disable your virtual machine for specific external interrupts, set the appropriate mask in control register 0.

How External Interrupt Handling Works

To understand how to use CMS facilities to handle your own interrupts, it helps to understand how external interrupt processing works—here is an overview (if you need more information, see *z/VM: ESA/XC Principles of Operation*.

1. **The CMS external FLIH saves status information** — When an external interrupt is reflected by CP to a virtual machine, the CMS external first-level interrupt handler (also called the external FLIH) performs the following processing:

    a. Saves information about the state of the interrupted program (for example, register contents).

    b. Captures the interrupt status information and stores it in an area mapped by the EXTUAREA mapping macro. (See "Entry and Exit Linkage" on page 177 for information on the format and location of this area.)

2. **The FLIH calls a SLIH** — After the first-level interrupt handler (FLIH) saves status information, it looks for the appropriate second-level interrupt handler (also called a SLIH). There are three types of SLIHs (code-specific handlers, a user-defined default handler, and the CMS default handler), and the FLIH looks for them as follows:

    a. **A code-specific handler** — First, the FLIH looks for a handler that was defined for the specific interrupt code.

        i) If a dummy handler is defined, the FLIH posts the ECB and returns control to the program that was interrupted first. If the program had issued the WAITECB macro, the wait state is cleared and processing can resume.

        ii) If a code-specific handler routine is defined, the FLIH invokes it in the addressing mode that was in effect at the time the HNDEXT macro that created the handler was issued. Upon return, the FLIH checks the return code in register 15.

        - If the return code is 8, the FLIH passes control to a user-defined default handler, which also receives the same interrupt information. If there is no user-defined default handler, the FLIH passes control to the CMS default handler.
        - If the return code is 4 or something other than 0 or 8, no ECB posting is required, and control is returned to the interrupted program.
        - If the return code is 0, the FLIH posts an ECB as specified by the handler routine (OS or VSE format). Control is returned to the program that was interrupted first. If the program had issued the WAITECB macro, the wait state is cleared and processing can resume.

    b. **The user-defined default handler** — If the user has not defined a handler for the specific code, or if the specific handler issues a return code of 8, the FLIH looks for a user-defined default handler.

Upon return from the user-defined default handler, the FLIH again checks the return code in register 15.

- If the return code is 8, the FLIH passes control to the CMS default handler.
- If the return code is 4 or something other than 0 or 8, no ECB posting is required, and control is returned to the interrupted program.
- If the return code is 0, the FLIH posts an ECB as specified by the handler routine (OS or VSE format). Control is returned to the program that was interrupted first. If the program had issued the WAITECB macro, the wait state is cleared and processing can resume.

c. **The CMS default handler —** If the FLIH cannot find a user-defined default handler, or if the user-defined handler issues a return code of 8, the FLIH calls the CMS default external interrupt handler.

The CMS Default External Interrupt Handler

CMS has a default interrupt handler that processes external interrupts that do not have specific handlers. The CMS default handler returns the interrupt code and issues a message to your terminal that asks if it should (a) ignore the interrupt and resume processing or (b) pass control to the CMS abend processing routines.

By contrast, previous releases of CMS used to place the virtual machine into the DEBUG environment when no interrupt handler was defined.

Handling Specific External Interrupts

Using HNDEXT, you can create a specific routine for handling external interrupts or you can create a 'dummy' handler to specify that the first-level interrupt handler post an ECB for specific interrupts.

In addition to any processing your handler does, it must store a return code in register 15. A return code of:

- 0 specifies normal completion. If the ECB parameter was specified, the first-level handler posts it.
- 4 instructs the first-level interrupt handler to not post an ECB.
- 8 instructs the first-level interrupt handler to call the user-defined default handler if one exists, or the CMS default handler.

Note: The HNDEXT SET function cannot define handlers for codes that already have handlers. To define a new handler for an interrupt code, you must clear the existing one and then define a new one.

Defining a Dummy Handler

To specify that the first-level external interrupt handler post an OS-type ECB and not pass control to a second-level handling routine, code the HNDEXT macro as follows:

```
HNDEXT SET,'DUMMY',CODE=extcode,ECB=(ecbaddr,OS)
```

When the external interrupt first-level handler detects the interrupt specified by CODE=extcode, it will post the ECB specified by the ECB parameter. Use the WAITECB macro, at the point in your program where you want to be notified of the external interrupt. See "Using the WAITECB Macro" on page 178 for more information on the WAITECB macro.

Defining a Specific Interrupt Handler

Example — No ECB Processing

To specify that the first-level external interrupt handler pass control to a second-level handling routine at address rtnaddr, code the HNDEXT macro as follows:

```
HNDEXT SET,rtnaddr,CODE=extcode
```

The CODE parameter of HNDEXT specifies the external interrupt code you wish to handle. Codes may be specified in betweenX'0001' to X'FFFE'.

Note: If you specify an external interrupt code of X'0000', a default handler is created to handle all external interrupts that do not have specific interrupt handlers. Also note that CMS uses the X'4000' external interrupt code to process IUCV interrupts.

Example — Optional ECB Posting

To specify that a first-level external interrupt handler can optionally post an ECB and pass control to a second-level handling routine at address rtnaddr, code the HNDEXT macro as follows:

```
HNDEXT SET,rtnaddr,CODE=extcode,ECB=ecbaddr
```

When the routine terminates it can issue a return code of 0 in register 15 to indicate posting of the ECB specified by the ECB parameter. A return code of 4 in register 15 would omit posting of the ECB.

Entry and Exit Linkage

You are responsible for providing the proper entry and exit linkage for your interrupt handling routine. When your program receives control, the register contents are as follows:

Register	**Contents**
0	Address of the user word specified on the HNDEXT macro.
1	Address of the area containing the state of the machine at the time of interrupt. The EXTUAREA DSECT maps this area; for more detailed information, see the *z/VM: CMS Macros and Functions Reference*.
2-11	Unspecified
12	Handling routine entry address
13	A pointer to the user save area (label UAREA) within the EXTUAREA
14	Return address
15	Handling routine entry address.

Your routine must return control to the address in register 14, and must store one of the following return codes in register 15.

- Zero (0) — Indicates the second-level handler is through handling the interruption and the first-level handler should post the ECB (if one was specified).
- Four (4) — Indicates the second-level handler has completed, and the first-level handler should **not** post the ECB.
- Eight (8) — Indicates that the interrupt is passed to a user-specified default handler, if one is defined. If a user-specified default handler does not exist, then the system default handler is invoked to handle the interrupt.

Address Translation Mode Consideration

In an XC virtual machine, all interrupt handlers always receive control in primary space (address translation) mode and always must return control to CMS in primary space mode.

Using the WAITECB Macro

ECBs, event control blocks, are standard mechanisms used to synchronize multiple events. The process of turning on the *event complete* bit is referred to as *posting* the ECB. The WAITECB macro suspends processing until a specific ECB or the ECBs in a list have been posted (this is called *waiting on an ECB*).

ECBs are fullwords and have the following OS or VSE format:

OS Format

```
   bit 0      WAIT bit
   bit 1      Event completed bit
   bit 2-31   Completion code
```

VSE Format

```
byte 0-1     Reserved
byte 2
   bit 0     Traffic bit
   bit 1-7   Reserved
byte 3       Reserved
```

Example 1

To wait on an OS format (the bit tested for "event completed" is byte 0 bit 1) ECB at address ECB1, code:

```
         WAITECB ECB=ECB1
```

Example 2

To wait on four VSE format ECBs at addresses ECB1, ECB2, ECB3, and ECB4, (the bit tested for event complete is byte 2 bit 0), code:

```
         WAITECB 4,ECBLIST=LISTADDR,FORMAT=VSE
.
.
LISTADDR DS     0F
         DC     A(ECB1)
         DC     A(ECB2)
         DC     A(ECB3)
         DC     A(ECB4)
FENCE    DC     X'FF'
.
.
ECB1     DC     F'0'
ECB2     DC     F'0'
ECB3     DC     F'0'
ECB4     DC     F'0'
```

Note that if FORMAT=VSE is specified:

- With ECBLIST, the byte following the last fullword of the list must be nonzero.
- The ECBs must reside below 16MB.

Example 3

To wait on two OS format ECBs at addresses ECB1 and ECB2, code:

```
         WAITECB 2,ECBLIST=LISTADDR
.
.
```

```
LISTADDR DS      0F
         DC      A(ECB1)
         DC      A(ECB2+X'80000000')
.
.
ECB1     DC      F'0'
ECB2     DC      F'0'
```

Note: If FORMAT=OS is specified (or defaulted) with ECBLIST, the high-order bit (bit 0) in the last fullword of the list must be set to 1.

Attention: If you specify a count larger than the number of ECBs in the ECBLIST, execution of this macro results in a permanent wait.

Creating Your Own Default Handler

To use the HNDEXT macro to create your own default external interrupt handler, omit the CODE parameter (or specify code asX'0000').

Example

To define your own default handler at address MYDEF, code the HNDEXT macro as follows:

```
HNDEXT SET,MYDEF
```

Deleting an External Interrupt Handler

Use HNDEXT CLR to delete an external interrupt handler.

Example 1

To delete a handler for a specific code (for example, X'40'), code:

```
HNDEXT CLR,40
```

Example 2

To delete handlers for the interrupt codes specified at CLRADDR, code:

```
         HNDEXT CLR,CLRLIST=CLRADDR
         .
         .
         .
         DS    0F
CLRADDR  DC CL4'0040'
         DC CL4'0050'
         DC CL4'0060'
         DC 8X'FF'
```

Notes:

1. The list of external interrupt codes you want to clear should end with an 8-byte fence of X'FF'.
2. Interrupt handling routines should not issue HNDEXT CLR.
3. If no codes are entered to be cleared, then the user-defined default handler is cleared.

Handling I/O Interrupts

I/O interrupts provide a means for your virtual machine to communicate with I/O devices. CMS has its own interrupt handling routines for all of the I/O devices it supports. Unless your program has special needs (for example, it does I/O to a device that CMS does not support) CMS default handlers should be sufficient.

If you do need to define your own interrupt handling routines, CMS provides a macro interface to help you. Use the CONSOLE macro to create exit routines for 3270 display devices (see "Handling Console Interrupts" on page 84 for details) and use HNDIO for other devices.

I/O Interrupt Handling — An Overview

After a program issues an I/O operation to a specific device, an interrupt is returned from the device indicating the status of the I/O operation. CP processes the interrupt first: it converts the results into a format your virtual machine can understand, and then calls the CMS I/O first-level interrupt handler (FLIH). The FLIH gets control in primary space mode and returns control in the same mode as when the interrupt occurred.

The CMS I/O FLIH saves information about (a) the state of the program that was running at the time the interrupt occurred and (b) the interrupt itself. The FLIH then looks for a second-level handler (SLIH) to process the interrupt. If a SLIH exists, the FLIH calls it. The SLIH processes the interrupt and returns control back to the FLIH, which will then return control to the program that was running when the interrupt occurred.

When you define your interrupt handler, you can specify that interrupts be handled in an alternate way. The FLIH will mark the interrupt as having occurred and will return. The SLIH will not be called until a later time determined by your program.

The following sections describe FLIHs and SLIHs in more detail and also describe how you can define your own interrupt processing routines.

Note: I/O operations initiated by some forms of the DIAGNOSE instruction do not produce I/O interruptions and are not trapped by HNDIO or HNDINT. If the I/O operation initiated by DIAGNOSE does produce I/O interrupts, then HNDIO will trap the interrupts if the device has been specified for HNDIO.

Enabling and Disabling I/O Interrupts

To enable or disable your virtual machine for all I/O interrupts, you can use the ENABLE macro (see "Manipulating the PSW Interrupt Mask" on page 173). To enable or disable your virtual machine for specific I/O interrupts, set the appropriate mask in control register 6.

First-Level I/O Interrupt Processing

As just mentioned, the first-level I/O interrupt handler (FLIH) processes reflected interrupts from CP, saves information about the state of the interrupted program, and captures the interrupt status information relative to a XA or XC virtual machine. After the FLIH saves information about the interrupt, it looks for a second-level handler (SLIH) for the device that caused the interrupt. In general, the FLIH looks for a SLIH as follows:

1. The FLIH determines if a device path defined by the CONSOLE macro is waiting for the interrupt. If one is, the FLIH resets the appropriate wait bits and restarts the program (in this case, the console facility).
2. If a defined console path was not waiting for an interrupt, the FLIH looks for handler routines in the following order:
   a. Interrupt handling routines defined by HNDIO or HNDINT.
   b. OS/MVS STAX exit routines (for attention interrupts only).
   c. Exit routines defined by the CONSOLE macro. If the last path that performed I/O defined an exit routine, that exit routine gets control. If no path did I/O, the FLIH determines if the last path opened specified an exit routine. If it did, the FLIH passes control to it.

   Once fullscreen mode has been established using the CONSOLE macro, only attention interrupts will be passed to exits for the virtual console. Other interrupts (such as unsolicited device ends) may be passed to exits defined for dedicated 3270 devices.
3. If the interrupt is for the virtual console and no handler is found by the FLIH, then CMS performs standard processing for the virtual console.

4. Finally, if the FLIH still cannot find a handler, the interrupt is ignored and control is returned to the routine that was executing when the interrupt occurred.

Second-Level I/O Interrupt Processing

CMS provides its own interrupt handling facilities for all devices it supports. Unless your program has special needs (for example, it performs its own I/O operations) or is using a device that CMS does not support, CMS interrupt handling facilities should be adequate. If you need to define your own interrupt handlers, you can use the HNDINT and HNDIO macros to do so or, use the CONSOLE macro exits for the virtual console or dedicated 3270 devices. Some applications still use the OS/MVS STAX macro for defining an exit routine to handle attention interrupts for the virtual console, but this is not a CMS preferred interface.

Note: The CONSOLE macro provides its own facilities for handling console interrupts. You should not define a CONSOLE exit and an HNDIO handler for the same device; however, if for some reason there is a CONSOLE exit and an HNDIO handler routine for the same device, the HNDIO routine overrides the CONSOLE exit unless the interrupt is expected. If an OS/MVS STAX routine is still in effect and there is a CONSOLE exit for the same device, the CONSOLE exit will not get control for unsolicited interrupts until the STAX routine is cleared.

While existing programs can continue to use the HNDINT macro, new programs should use HNDIO. HNDIO provides the same functions as HNDINT, in addition, it allows a programmer to:

- Obtain complete I/O status for XC and XA virtual machines (the INTBLOK parameter)
- Specify whether the interrupt handler is to be cleared at end-of-command (the KEEP parameter)
- Specify whether the handler *survives* abend processing (the SYSTEM parameter)
- Specify whether the handler survives machine check processing for the detaching or redefining of the device (the PERSIST parameter)
- Specify an error return address (the ERROR parameter)
- Specify various macro formats (the MF parameter).

Defining an I/O Interrupt Handling Procedure

If you need to define an interrupt handler for any device other than a display device, use the HNDIO macro. (Use the CONSOLE macro to define handlers for 3270 displays doing full-screen I/O.) Here are some things to consider when you use HNDIO to define your own interrupt handlers:

- Is the handler for a program that runs in a 370 virtual machine or in an XC or XA virtual machine?

  The format of the interrupt information you receive varies according to the mode of the virtual machine. To receive complete I/O status in an XC or XA virtual machine, you must specify the INTBLOK parameter and provide a storage area where the CMS I/O first-level handler can store information. You can use the INTBLOK mapping macro to map this area.

  Because having an INTBLOK area has no adverse effects on programs running in a 370 virtual machine all of the examples in the next section define one.

  Note: Only CMS levels prior to CMS Level 12 can execute in a 370 virtual machine.
- Should the handler survive end-of-command and abend processing?

  By default, CMS clears user-defined interrupt handlers at end-of-command or when it (CMS) performs abend processing. The SYSTEM and KEEP parameters of the HNDIO macro let you save interrupt handlers across end-of-command and abend processing.

  Note: If you specify SYSTEM=YES and KEEP=YES for an interrupt handler, you must make sure that the interrupt handler itself survives abend and end-of-command processing. To make sure the handler survives, you can place the entry address of the routine (rtnaddr) in a nucleus extension, or you can allocate the storage for the handler from a global subpool for which you have specified SYSTEM=YES.
- Are there times other than the ones mentioned above, when handlers will be cleared?

In an XC or XA virtual machine, if a machine check interrupt is received because of a device being detached or redefined, then the normal system action is to clear all user defined I/O interrupt handlers associated with that device. However, this action can be overridden if the interrupt handler has been specified with PERSIST=YES, which allows the exit clearing to be ignored when the machine check interrupt is processed.

- When should the handler be notified of the interrupt?

  You have two options:

  1. You can instruct CMS to call your interrupt handler when the interrupt comes in (this is the default, NOTIFY=ASAP).
  2. You can instruct CMS to call your interrupt handler only when you want it to (to do this specify the WAITD macro at the point in your program where you want to be notified and specify NOTIFY=WAIT on the HNDIO macro).

Handling Interrupts — Examples

For all of the following examples, assume that you are creating a handler for the device named DEVO (DEVNAME=DEVO), that the device number of DEVO is 990 (DEVICE=990), and that the exit routine begins at label ZEESLIH (EXIT=ZEESLIH). Also note that all of the following examples define an INTBLOK, the storage for which you must provide.

Immediate Notification

The NOTIFY parameter specifies whether you want notification and handling of the interrupt to be immediate (when the interrupt comes in) or synchronous (only when your program issues a WAITD macro). NOTIFY=ASAP (immediate notification) is the default value.

To define a handler that gets notified immediately, code:

```
HNDDEVO  HNDIO SET,DEVNAME=DEVO,EXIT=ZEESLIH,DEVICE=990,
               NOTIFY=ASAP,INTBLOK=((R8),INBLKSZ)
```

Whenever it receives an interrupt from DEVO, CMS immediately calls the handler at ZEESLIH.

Synchronizing Interrupt Processing

To instruct CMS to delay notification until your program is ready, specify NOTIFY=WAIT on the HNDIO macro:

```
HNDDEVO  HNDIO SET,DEVNAME=DEVO,EXIT=ZEESLIH,DEVICE=990,
               NOTIFY=WAIT,INTBLOK=((R8),INBLKSZ)
```

To define the point in your program where you want to be notified of the interrupt, code the WAITD macro. If the interrupt has already come in, CMS calls your handler immediately. If the interrupt has not come in, CMS suspends the execution of your program until it does. When your handler finishes, CMS returns control to the instruction following the WAITD macro.

Saving the Handler Across Abend and End-of-Command Processing

Use the KEEP and SYSTEM parameters to specify that a handler is to survive end-of-command and abend processing. Note, however, that if you specify either KEEP=YES or SYSTEM=YES, you must also make sure that the storage that contains the handler survives.

The following example creates a global subpool that survives abend processing and then uses the storage to create an interrupt handler.

```
         .
         .
         .
         SUBPOOL CREATE,NAME='SLIPOOL',TYPE=GLOBAL,SYSTEM=YES,         X
               KEY=NUCLEUS
*
         LA    R3,HNDLRLEN        Length of interrupt handler
         CMSSTOR OBTAIN,BYTES=(R3),SUBPOOL=('SLIPOOL',GLOBAL)
*
         LR    R4,R1              Save for exit definition
         LR    R2,R1              Where to put the Handler
         LR    R3,R0              Size of the handler
         LA    R0,MYHNDLR         Addr of the handler in this program
         LR    R1,R3              Size of the handler
         MVCL  R2,R0              Copy into global storage
*
         LA    R3,INBLKSZ         Length of our INTBLK
         CMSSTOR OBTAIN,BYTES=(R3),SUBPOOL=('SLIPOOL',GLOBAL)
         LR    R5,R1              Save INTBLK Address
*
         HNDIO SET,DEVNAME=DEV0,EXIT=(R4),DEVICE=990,NOTIFY=ASAP,      X
               INTBLOK=((R5),(R3)),KEEP=YES,SYSTEM=YES
         .
         .
         .
MYHNDLR  EQU   *   Start of interrupt handler to survive abend
*                  processing and end-of-command.  It must have NO
*                  relocatable addressing constants when moved
*                  into free storage.
         .
         .
         BR    R14
*
HNDLRLEN EQU   *-MYHNDLR   Size of the above interrupt handler
```

Using a Dummy Handler — Ignoring Interrupts

A ‘dummy’ handler is one that has no interrupt handling code defined. For example, to ignore interrupts from device 990 you could code:

```
HNDDEVO  HNDIO SET,DEVNAME=DEVO,EXIT=0,DEVICE=990,NOTIFY=ASAP
```

or, because EXIT=0 and NOTIFY=ASAP are defaults

```
HNDDEVO  HNDIO SET,DEVNAME=DEVO,DEVICE=990
```

Using a Dummy Handler — Notification Only

If you do not want to process interrupts from device 990 but you do want to know when they come in, you could code:

```
HNDDEVO  HNDIO SET,DEVNAME=DEVO,DEVICE=990,NOTIFY=WAIT
```

At the point in your program where you want to receive notification, issue the WAITD macro.

Waiting For Unsolicited Interrupts

There may be times when you want to use the WAITD macro without defining an interrupt handler. For example, assume you wanted a program that would handle unsolicited interrupts from your reader, but remained in a wait state otherwise. Your program could issue WAITD RDR; when an interrupt from your reader comes in, it will ‘wake up’ your program, which can then process the interrupt. When your program finishes handling the interrupt, it can re-issue the WAITD RDR thus returning to a wait state.

Interrupt Handler Conventions

When your interrupt handler routine receives control, all I/O interruptions and external interruptions are disabled. Your interrupt handler should not enable I/O or external interrupts, perform any I/O operations, or issue a HNDIO CLR for the device associated with the handling routine.

You must provide the proper entry and exit linkage for your interrupt handlers. CMS calls second-level interrupt handlers in the same addressing mode as the program that issues the HNDINT or HNDIO macro.

Address Translation Mode Consideration

In an XC virtual machine, all interrupt handlers always receive control in primary space (address translation) mode and always must return control to CMS in primary space mode.

Linkage Entry Conventions

For the most part, HNDIO, HNDINT, and the second-level routines they define use the same linkage conventions. The only exception is that second-level routines defined by HNDIO use register 5 to point to the INTBLOK and register 6 to contain the UWORD (if one was specified). For second-level interrupt handlers created by HNDINT, CMS zeros out registers 5 and 6.

Register	Contents
0-1	Contains the I/O old PSW. For an XA or XC virtual machine, registers 0-1 contain a re-constructed PSW in BC format. XA or XC mode programs can extract the real I/O old PSW from the INTBLOK.
2-3	Contains the Channel Status Word (CSW). For an XA or XC virtual machine, registers 2-3 contain a re-constructed CSW. The XA or XC mode program can extract the real I/O status from the INTBLOK's IRB content.
4	Contains the address of the device that caused the interrupt. For an XA or XC virtual machine, this address is extracted from the I/O interrupt code at location X'B8'. The XA or XC mode program may extract more complete device information from the INTBLOK.
5	Points to the INTBLOK that the HNDIO macro specifies. For an XA or XC virtual machine, the INTBLOK contains 370-XA interrupt information, such as the I/O old PSW, the I/O interrupt code, and the IRB. For 370 virtual machines, the INTBLOK contains CSW information in the appropriate IRB fields. If you do not specify the INTBLOK parameter, the contents of R5 are 0.
6	Contains the user word specified on the UWORD parameter of the HNDIO macro. If the UWORD parameter was not specified, then register 6 is set to zero.
7-11	Unspecified.
12	Contains the entry address of the interrupt handling routine.
13	Points to a 24-word save area provided by the first-level I/O interrupt handler.
14	Contains the return address.
15	Contains the entry address of the interrupt handling routine.

See *z/VM: CMS Macros and Functions Reference* for an expansion of the INTBLOK macroinstruction (INTBLOK DSECT).

Note: To obtain interrupt information in an XA or XC virtual machine, the FLIH issues a TSCH (test subchannel) instruction. If the TSCH fails, the INTSTAT field of the INTBLOK control block is set to X'FF's and the related IRB information is invalid for this interrupt.

Linkage Exit Conventions

Interrupt handling routines must return control to the address in register 14. They must also store a return code in register 15 to indicate whether processing is complete. A zero (0) in register 15 indicates the second-level handler is through handling the interruption; a nonzero return code indicates that the second-level handler expects another interrupt before processing completes. (For information on the CONSOLE exit routine entry conditions, see Chapter 8, “Console and Terminal I/O,” on page 83.)

Clearing an Interrupt Handler

To delete an interrupt handler, use HNDIO CLR.

Note: Do not issue HNDIO CLR from within an interrupt handler.

Example — Clearing a Specific Interrupt Handler

The following example shows how to use the HNDIO CLR to clear the interrupt handler for DSK0:

```
HNDIO CLR,DSK0
```

Example — Clearing Interrupt Handlers for a List of Devices

Use the CLRLIST parameter to clear interrupt handlers for a list of devices. Specify each device in the list as a four-character symbolic name. To make sure that only the specified device interrupt handlers are cleared, end the list with a fence (8X'FF').

If your routine attempts to clear a handler for an unspecified device, the invalid device name is returned in register 1. Handlers for devices prior to the invalid device are cleared; handlers for devices following the invalid device are not.

The following example shows how to use the CLRLIST parameter to clear handling routines for DSK1, DSK2, DSK3, and DSK4:

```
         HNDIO CLR,CLRLIST=LISTADDR
         .
         .
         .
LISTADDR DS   0H
         DC   CL4'DSK1'
         DC   CL4'DSK2'
         DC   CL4'DSK3'
         DC   CL4'DSK4'
         DC   8X'FF'
```

Second-Level Handler Returns to First-Level

After a handler routine processes an interrupt, control is returned to the FLIH. The FLIH then examines the I/O wait bit in the I/O old PSW to determine if a program was waiting for the interrupt. (Programs can issue the WAITD macro to set the I/O wait bit on and thus halt execution until the interrupt arrives.)

1. If the I/O wait bit is off, the FLIH determines if the program intended to wait for the interrupt but had not yet issued the WAITD macro. (In other words, if the handler routine for the device specified NOTIFY=WAIT and the I/O wait bit is off, then the interrupt arrived **before** the program had a chance to issue a WAITD macro.)
   a. If the handler routine did not specify NOTIFY=WAIT, the FLIH restarts the program that was executing when the I/O interrupt occurred.
   b. If the handler routine specified NOTIFY=WAIT, the FLIH saves information about the interrupt; when the program issues the WAITD, the handler routine is immediately invoked. After saving the status information, the FLIH then restarts the program that was executing when the I/O interrupt occurred.
2. If the I/O wait bit is on, the FLIH determines if a WAITD had been issued for the device that caused the interrupt.

a. If a program did issue a WAITD for the device that caused the interrupt, the FLIH resets the WAITD indicator and the wait bit in the I/O old PSW and restarts the interrupt process: this returns control to the issuer of the WAITD.
b. If no WAITD was issued for the device that caused the interrupt, then the user is waiting for an interrupt from another device. The FLIH does not restart the program; rather, the program remains in a wait state until the anticipated interruption occurs.

SVC Interrupts

Use the HNDSVC macro to set or clear routines that trap interrupts caused by specific supervisor call (SVC) instructions.

Creating SVC Handlers

Use HNDSVC SET to create SVC handlers.

Example 1

To define an SVC handler for SVC 10 that begins at ADDR10, code the following:

```
HNDSVC SET,(10,ADDR10)
```

Example 2

To define (a) an SVC handler for SVC 10 that begins at ADDR10 and (b) an SVC handler for SVC 11 that begins at ADDR11, code the following:

```
HNDSVC SET,(10,ADDR10),(11,ADDR11)
```

Example 3

To define (a) an SVC handler for SVC 10 that begins at ADDR10, (b) an SVC handler for SVC 11 that begins at ADDR11, and specify that (c) CMS keep both of the user SVC handler definitions past end-of-command processing, code the following:

```
HNDSVC SET,(10,ADDR10),(11,ADDR11),KEEP=YES
```

Example 4

To define (a) an SVC handler for SVC 10 that begins at ADDR10, (b) an SVC handler for SVC 11 that begins at ADDR11, and specify that (c) CMS keep both of the user SVC handler definitions past end-of-command processing and (d) both handlers survive beyond abend processing, code the following:

```
HNDSVC SET,(10,ADDR10),(11,ADDR11),KEEP=YES,SYSTEM=YES
```

Note: The SYSTEM parameter, like KEEP, applies to all trap routines defined on an HNDSVC SET macro, not to individual trap routines.

Example 5

The UWORD parameter, unlike KEEP and SYSTEM can be specified for any individual SVC handler on a single HNDSVC SET macro invocation. For example, to define (a) an SVC handler for SVC 6 that begins at ADDR6 and passes the fullword contained at UWD6 and (b) an SVC handler for SVC 7 that begins at ADDR7, code the following:

```
HNDSVC SET,(6,ADDR6,UWORD=UWD6),(7,ADDR7)
```

HNDSVC Entry and Exit Linkage

You must provide the proper entry and exit linkage for your SVC handling routine. When your program receives control, the register contents are as follows:

Register
Contents

0-11
Remain the same as when the SVC was issued.

12
If the current addressing mode is AMODE 24, register 12 contains the SVC number in the high-order byte and a 3 byte address of the routine. If the addressing mode is AMODE 31, register 12 contains only the address of the SVC trap routine. For both addressing modes, the address of the SVC trap routine will always be in register 12 and the UWORD and SVC number can always be found in the HSVCSAVE pointed to by register 13.

13
The address of an HSVCSAVE save area.

14
The return address to the SVC handler.

15
Remains the same as when the SVC was issued.

When complete, your routine must return control to the address in register 14. You do not need to restore any registers. The registers are restored to the contents they held at the time the SVC was issued.

Address Translation Mode Consideration

In an XC virtual machine, all interrupt handlers always receive control in primary space (address translation) mode and always must return control to CMS in primary space mode.

Controlling the Addressing Mode of SVC Trap Routines

SVC trap routines receive control in the addressing mode of the program that issues the HNDSVC macro, not in the addressing mode of the program that issues the *trapped* SVC. The HNDSVC macro does not have an addressing mode parameter that allows the user to dynamically set the AMODE. Therefore, to change or set the addressing mode of an SVC trap routine, issue the AMODESW macro.

Deleting SVC Handlers

Use HNDSVC CLR to delete SVC handlers.

Example 1

To delete the handler for SVC 10, code

```
HNDSVC CLR,10
```

Example 2

To delete the handlers for SVC 10 and SVC 11, code

```
HNDSVC CLR,10,11
```

Note: SVC numbers 0 through 200 and 206 through 255 are valid.

Machine Check Interrupts

Machine-check interrupts provide a means of reporting equipment malfunctions. CP reflects machine checks to a virtual machine when it detects events that affect a virtual machine or to notify the virtual machine of damage it (CP) has detected. For example, CP presents a host paging error to a virtual machine as a storage-error machine-check since the virtual machine's storage is effectively damaged.

The CMS machine-check handler takes control when it receives a machine-check interrupt from CP. If possible, CMS responds to machine checks by returning control to the program that's running. Otherwise, CMS loads a disabled wait state PSW. To continue, you must re-IPL CMS.

One of the common causes of a machine check is a change in the I/O configuration of an XA or XC virtual machine. The CMS machine check handler will record the change, and if the change in I/O configuration causes a device to be detached or redefined at a new address (using a CP DETACH, CP DEFINE, or CP REDEFINE command), then I/O exits associated with that device will be cleared.

Data Space Machine Checks

An application can receive storage errors and I/O errors on a reference to data contained in an address space other than its own primary address space. This can be another user's virtual machine primary address space or a data space.

Storage Errors

When an uncorrected storage error is detected by CP, a machine check is reflected to the XC virtual machine. This type of error is typically caused by:

- Real error main storage
- Paging error on page-in of a page.

In these cases, the data that was in the page is lost. CMS checks the ASIT supplied with the machine check interrupt to determine if the storage error was in a data space. If it was, CMS issues a system abend with a CMS abend code of X'1F4'.

I/O Errors

When an I/O error is detected by CP while trying to do a page-out of a mapped page to its respective DASD slot, a machine check is reflected to the owning virtual machine. In this case, the data in storage is still valid. CMS issues a system abend with a CMS abend code of X'1F5'.

For more information on these machine checks, see the "Storage Error Notification for Access Register-Specified References" section of the "Using Data Spaces" chapter of the *z/VM: CMS Application Development Guide*.

Program Interrupt Handling

Program interrupts report exceptions and events that occur during execution of a program. When CP detects a program interrupt, it reflects that interrupt to CMS. CMS, in turn, stores information about the interrupt, and abends the program.

The information CMS stores depends on the mode of the virtual machine (370, XA, or XC). For 370 virtual machines, the interrupt code field and ILC field of the PSW contain interrupt information. XA and XC virtual machines have this information returned at assigned storage locations. In addition to the interrupt code and ILC information, XA and XC virtual machines also receive information about what caused the interrupt.

Note: Only CMS levels prior to CMS Level 12 can execute in a 370 virtual machine.

CMS First-Level Program Interrupt Processing

When the CMS first-level program interrupt handler (FLIH), receives control, it saves information about the program that was interrupted, it saves information about the program interrupt itself, and it abends the program.

The FLIH saves information about the program interrupt in a data area called the Extended Program Interruption Element (EPIE). This EPIE contains the interrupt information as required by a SPIE/ESPIE exit routine.

After saving information, CMS abends the program. As part of the abend process, CMS searches for user exits in the following order:

1. Exit routines defined by the OS/MVS SPIE or ESPIE macros.
2. Exit routines defined by the OS/MVS STAE or ESTAE macros. If a STAE/ESTAE is defined, the FLIH does some preparatory work and issues an MVS SVC 13 (ABEND) to invoke the MVS simulation routines that process the STAE/ESTAE exits. Control does not return to the FLIH.

   Note: Changes in OS simulation affects the way CMS manages OS/MVS resources. For more information on OS/MVS resource management, see "OS/MVS Resource Management" on page 338.
3. Exit routines defined by the CMS ABNEXIT macro (see "Creating Abend Exit Routines" on page 204 for more information about ABNEXIT).

Address Translation Mode Consideration

In an XC virtual machine, all interrupt handlers always receive control in primary space (address translation) mode and always must return control to CMS in primary space mode.

If a CMS system program was running when the interrupt occurred, CMS does not look for user-defined exits.

Defining Program Interrupt Handlers

Use the OS/MVS SPIE and ESPIE macros to define handler routines for specific interrupt codes. For more information on these macros, see the *MVS/XA Supervisor Services and Macro Instructions*.

Chapter 13. Nucleus Extensions and Commands

This chapter describes how to use the following macros:

- NUCEXT — declares, clears, renames, and queries nucleus extensions.
- ANCHOR — retrieves, declares, and deletes an anchor word.
- SUBCOM — defines, clears, and queries subcommand environments.
- IMMCMD — declares, clears, and queries immediate commands.

For information on how to use CMS commands to manage programs, see Chapter 15, "Program Packaging," on page 211.

Nucleus Extensions

A nucleus extension is a program in free storage that CMS treats as if it were a nucleus command. This means that, once loaded, the nucleus extension remains in storage; when the nucleus extension is called, CMS does not have to read it into storage from a disk.

The NUCEXT Macro

The NUCEXT macro defines, clears, and queries a nucleus extension and is independent of the PSW format.

The basic formats of NUCEXT are:

- NUCEXT SET — to declare a nucleus extension
- NUCEXT ANCHOR — to get the anchor pointer for the SCBLOCKs that describe the list of nucleus extension programs
- NUCEXT CLR — to delete a nucleus extension from the list of SCBLOCKs
- NUCEXT QUERY — to determine whether a nucleus extension is currently defined
- NUCEXT RENAME — to change a nucleus extension name.

Should Your Program Be a Nucleus Extension?

Defining a frequently used program as a nucleus extension improves the performance of the program — CMS does not need to continuously read the program in from disk. Programs that you might consider defining as nucleus extensions include:

- Programs that gather statistics.
- Programs that filter CMS commands.
- Programs to be invoked during CMS end-of-command processing or abend processing.
- Interrupt handlers.

  Note: Interrupt handlers that are loaded as nucleus extensions are not allowed to request OS/MVS services.

Defining Nucleus Extensions

Use the NUCEXT SET macro to declare a nucleus extension.

Example 1

To declare a nucleus extension named MYNUC at entry point address PROGADDR, code the NUCEXT macro as follows

```
NUCEXT SET,NAME='MYNUC',ENTRY=PROGADDR
```

Unless AMODE is specified on the NUCEXT macro, the addressing mode of the nucleus extension is the same as the addressing mode of the program that creates it.

Example 2 — Specifying an Addressing Mode

To specify that MYNUC run in 31-bit mode, code the NUCEXT macro as follows:

```
NUCEXT SET,NAME='MYNUC',ENTRY=PROGADDR,AMODE=31
```

Note that MYNUC runs in 31-bit mode in an XA or XC virtual machine, but runs in 24-bit addressing mode in a 370 virtual machine.

Example 3 — End-of-Command Nucleus Extensions

To specify that MYNUC receive control during CMS end-of-command processing, code the NUCEXT macro as follows:

```
NUCEXT SET,NAME='MYNUC',ENTRY=PROGADDR,ENDCMD=YES
```

Example 4 — Immediate Command Nucleus Extensions

To define MYNUC as an immediate command, code the NUCEXT macro as follows:

```
NUCEXT SET,NAME='MYNUC',ENTRY=PROGADDR,IMMCMD=YES
```

Example 5 — Service Call Nucleus Extensions

To specify that MYNUC receive control during abend processing or when the NUCXDROP command is issued, code the NUCEXT macro as follows:

```
NUCEXT SET,NAME='MYNUC',ENTRY=PROGADDR,SERVICE=YES
```

Specifying Values at Execution Time

The values of the SYSTEM, SERVICE, ENDCMD, IMMCMD and KEY parameters may be determined at execution time by using the (reg) or (addr,mask) forms. For example, the sequence:

```
L      R2,SERVOP    Get value of service parameter
NUCEXT SET,MF=(L,NUCEXTP),SERVICE=(R2)
```

tests the value of register 2 to determine the value of the SERVICE parameter. If register 2 contains zero, then SERVICE=NO is set. Otherwise, SERVICE=YES is set.

The (addr,mask) form allows a bit flag in storage to be used for setting these parameters. The macro call

```
NUCEXT SET,MF=(L,NUCEXTP),SERVICE=(FLAGS,SERVFLAG)
```

will cause the bit in the byte at address FLAGS (at the offset determined by SERVFLAG) to be tested (using the instruction TM FLAGS,SERVFLAG). If this bit is zero, then SERVICE=NO is used, otherwise SERVICE=YES is used.

Other Parameters You Can Specify on NUCEXT

The following list briefly describes some of the other parameters of the NUCEXT macro you can use (for complete details about the NUCEXT macro, see the *z/VM: CMS Macros and Functions Reference*):

- UWORD — specifies an optional user word stored in the SCBWKWRD of the SCBLOCK.
- UFLAGS — specifies an optional one byte parameter in the SCBLOCK.

- AMODE — specifies the addressing mode in which the nucleus extension is entered.
- INTTYPE — specifies the PSW interrupt mask the CMS SVC interrupt handler uses when invoking the nucleus extension.
- ORIGIN — specifies the location and length (in bytes) of the program in virtual storage.
- KEY — specifies the storage key (NUCLEUS or USER).
- SYSTEM — specifies whether the nucleus extension survives CMS abend processing.

Defining Nucleus Extensions in Logical Saved Segments

You can also define nucleus extensions by loading the module or text files in a logical saved segment. When the logical saved segment containing the nucleus extension is loaded, the nucleus extension is activated. If a nucleus extension with the same name already exists, it is overridden in a stack-like manner.

Example 1

To create a nucleus extension named MYNUC, add the following record in the logical segment definition file for the MYSEG logical saved segment:

```
MODULE MYNUC * (SYSTEM)
```

MYNUC is entered in key zero and disabled for interrupts. The addressing mode is determined by the way you code the program and the location in the virtual machine where the logical saved segment is loaded. To save MYNUC across an abend, you must specify the SYSTEM option on the SEGMENT LOAD command or code the SYSTEM parameter on the SEGMENT LOAD macro when you load the logical saved segment

Example 2 — End-of-Command Nucleus Extensions

To specify that MYNUC receives control during CMS end-of-command processing, add the following record in the MYSEG logical segment definition file:

```
MODULE MYNUC * (ENDCMD)
```

Example 3 — Immediate Command Nucleus Extensions

To define MYNUC as an immediate command, add the following record in the MYSEG logical segment definition file:

```
MODULE MYNUC * (IMMCMD)
```

Example 4 — Service Call Nucleus Extensions

To specify that MYNUC receive control during abend processing or when the NUCXDROP command is issued, add the following record in the MYSEG logical segment definition file:

```
MODULE MYNUC * (SERVICE)
```

For more information on creating nucleus extensions in logical saved segments, see *z/VM: Saved Segments Planning and Administration*.

Nucleus Extension Entry Conditions

When CMS calls a nucleus extension, the register contents are:

Register
  Contents

R0
  Address of extended parameter list (if one was provided by the caller).

R1
Address of the command name (and the tokenized parameter list).

R2
Address of an SCBLOCK that contains information about the nucleus extension.

R12
Entry point address.

R13
User save area mapped by the USERSAVE macro. Note that the USECTYP field of the user save area contains call type information. For 24-bit applications, this information is also found in the high-order byte of register 1. If the nucleus extension is called during end-of-command processing (ENDCMD=YES), the call type is X'FE'. If the nucleus extension is called during abend processing (SERVICE=YES), the call type is X'FF'.

R14
Return address.

R15
Entry point address.

This is the standard entry point convention except that R2 points to the SCBLOCK.

Obtaining the Anchor Point of the SCBLOCK List

Use NUCEXT ANCHOR to obtain the anchor pointer for the list of SCBLOCKs that describe the list of current nucleus extension programs. The pointer to the first entry in the NUCEXT list of SCBLOCKs is returned in register 1.

For example, to obtain the anchor pointer for nucleus extensions, code

```
NUCEXT ANCHOR
```

Note: The ANCHOR option requires a read/write parameter list; therefore, use the Execute form of the macro (MF=(E,addr)) if you require reentrant code.

Deleting Nucleus Extensions

Use NUCEXT CLR to delete a nucleus extension from the list of SCBLOCKs that describe the current list of nucleus extension programs. For example, to clear the nucleus extension named MYNUC, code

```
NUCEXT CLR,NAME='MYNUC'
```

Obtaining Information about Nucleus Extensions

Use NUCEXT QUERY to query whether a nucleus extension is currently defined. If the nucleus extension you specify is defined, NUCEXT stores in register 1 the address of the SCBLOCK. If the nucleus extension is not defined, NUCEXT stores in register 15 a return code of 1. For example, to determine if the nucleus extension named MYNUC is defined, code

```
NUCEXT QUERY,NAME='MYNUC'
```

Note: The QUERY option requires a read/write parameter list; therefore, use the Execute form of the macro (MF=(E,addr)) if you require reentrant code.

Renaming Nucleus Extensions

Use NUCEXT RENAME to change a nucleus extension name. NUCEXT RENAME changes the name field of an SCBLOCK for a nucleus extension. For example, to rename the nucleus extension MYNUC to TESTNUC, code

```
NUCEXT RENAME,NAME='MYNUC',NEW='TESTNUC'
```

If the nucleus extension is not defined, NUCEXT stores a return code of 28 in register 15.

ANCHOR Words

ANCHOR provides quick access to an anchor word, which is a fullword that points to one or more control blocks allocated in free storage by a program. This avoids the overhead of obtaining dynamic storage each time the program is invoked. This anchor word persists between calls to the program and persists after an abend occurs.

Before using ANCHOR, you must request an anchor identifier from IBM. This is necessary to ensure that your identifier is unique among all programs using the anchor facility.

To request your anchor identifier, complete the *ANCHOR Identifier Registration Form* included at the end of this book, and mail it to IBM. IBM will assign you an anchor identifier and notify you by mail.

The ANCHOR Macro

The ANCHOR macro sets, queries, and clears a fullword that can be used by a program to save the address of its data between invocations.

The basic formats of ANCHOR are:

- ANCHOR QUERY — to retrieve the contents of a previously set anchor word.
- ANCHOR SET — to declare an anchor word.
- ANCHOR CLEAR — to delete an anchor word.

What Types of Programs Should Use the ANCHOR Macro?

Only programs with critical performance needs, such as programs called multiple times per second, should use the ANCHOR macro. The Anchor facility keeps a list of 16 anchor words and their associated anchor identifiers. The number of anchor slots is limited to 16 for two reasons: (1) to reduce the time to search the list and (2) because it is unlikely that more than 16 performance-critical applications would be competing for execution at the same time during a CMS session.

Programs that do not have critical performance needs should use a nucleus extension to keep their anchor word.

Defining ANCHOR Words

Use the QUERY parameter to retrieve the contents of a previously set anchor word.

Example 1— To query an Anchor word named ABC, code the ANCHOR macro as

```
ANCHOR QUERY,IDENT=ABC,ERROR=(R2)
```

In this example, QUERY checks to see if an anchor slot has been assigned for ABC. If it has, QUERY returns the anchor word in register 1. If ABC is not found, the routine specified in the ERROR parameter is run.

Use the SET parameter to declare an Anchor word.

Example 2— To declare an Anchor word named ABC, code the ANCHOR macro as

```
ANCHOR SET,IDENT=ABC,ERROR=(R2),VALUE=(R7)
```

In this example, SET initializes the anchor word to the value specified in register 7. If VALUE is omitted, the anchor word is set to zero.

Use the CLEAR parameter to delete an anchor word from the list of anchor slots.

Example 3— To clear an anchor word named ABC, code the ANCHOR macro as

```
ANCHOR CLEAR,IDENT=ABC,ERROR=(R2)
```

In this example, CLEAR checks to see if an anchor slot has been assigned for ABC. If it has, CLEAR sets the anchor identifier and the anchor word to binary zeros. If ABC is not found, the routine specified in the ERROR parameter is executed.

Anchor Entry Conditions

When your program calls the anchor facility, the register contents are:

Register	**Contents**
R0	Your 3-character anchor identifier and a blank.
R1	During an ANCHOR SET, this register contains the anchor word data.
R14	Return address.
R15	Entry point address.

Note: Anchor support is not access register mode capable. Applications executing in an XC virtual machine must be sure to call ANCHOR only in primary space mode.

Using ANCHOR

The following is an example of ANCHOR usage. ANCHOR QUERY is usually the first invocation of the Anchor facility from an application program. If the anchor word has already been set, the program can simply continue with its processing. If the anchor word has not been set, then the program must initialize before processing can continue.

```
*
* Set up program linkage
*
         …
*
* Check to see if we already have our data area.
*
         ANCHOR QUERY,IDENT=XYZ,ERROR=INIT
*
* Reg 1 now points to our data area.
*
         B     CONTINUE           Begin execution of the main code
INIT     EQU   *                  First-time initialization
*
* Obtain storage for our data area and initialize it here
*

         …
*
* Initialize our anchor word.
* R2 contains the address of our data area.
*
         ANCHOR SET,IDENT=XYZ,VALUE=(R2),ERROR=NOANCHOR
         LR    R1,R2              Set up the base for our data area
CONTINUE EQU   *                  Main code starts here
         …
         BR    R14                Return to caller
*
* There was no room for an anchor word.
* Issue an error message and exit
*
NOANCHOR EQU   *                  Come here if no room for anchors
         …
         BR    R14                Return to caller
```

ANCHOR CLEAR will usually be invoked during cleanup after an abend. If the storage that the anchor word points to was allocated in a separate subpool, the cleanup routine could return the entire subpool at one time.

```
*
* Set up program linkage and check to see if we need to clean up.
*
         …
*
* Clear the anchor to our data area.
* If no anchor was set, we do not need to clear it.
*
         ANCHOR CLEAR,IDENT=XYZ,ERROR=RETURN
RETURN   EQU   *
         BR    R14
```

Subcommand Environments

A subcommand environment is one where a program defines to CMS a number of routines, or subcommands, that CMS can then recognize and invoke by name. For example, you can create a program named COLOR that, in turn, defines a number of subcommands named RED, YELLOW, and BLUE. While you run the COLOR program, CMS recognizes RED, YELLOW, and BLUE as subcommand names; when you enter a subcommand name, CMS calls the appropriate entry point in your program.

SUBCOM Macro

You can use the SUBCOM macro to define, clear, and obtain information about a subcommand environment. The SUBCOM macro supports 31-bit addressing and provides an interface to the SUBCOM function.

The basic formats of the SUBCOM macro are:

- SUBCOM SET — to indicate the start of a subcommand routine
- SUBCOM CLR — to delete a subcommand environment from a list of currently active subcommand environments
- SUBCOM QUERY — to determine whether a subcommand environment is currently defined
- SUBCOM ANCHOR — to obtain the anchor pointer for the list of SCBLOCKs that describe currently active subcommand environments.

Defining Subcommands

Use SUBCOM SET to declare a subcommand.

Example 1

To create a subcommand processor entry point named BLUE at address BLUEADDR, code the SUBCOM macro as follows:

```
SUBCOM SET,NAME='BLUE',ENTRY=BLUEADDR
```

Unless AMODE is specified on the SUBCOM macro, the addressing mode of the subcommand processor is the same as the addressing mode of the program that creates it.

Example 2

To specify that BLUE runs in 31-bit mode, code the SUBCOM macro as follows:

```
SUBCOM SET,NAME='BLUE',ENTRY=BLUEADDR,AMODE=31
```

Note that BLUE runs in 31-bit mode in an XA or XC virtual machine, but runs in 24-bit addressing mode in a 370 virtual machine.

Example 3 — Saving a Subcommand Processor Across an Abend

To specify that BLUE survive an abend, code:

```
SUBCOM SET,NAME='BLUE',ENTRY=BLUEADDR,SYSTEM=YES
```

Note that to survive an abend, a subcommand processor must reside in storage that will not be reclaimed during abend processing. (You can use the SUBPOOL macro to create storage subpools that survive abend processing, see "Example 5 — Saving Global Subpools Across Abends" on page 59 for details.)

You can also determine the value of the SYSTEM parameter at execution time by using SYSTEM=(reg) or SYSTEM=(addr,mask). For example, the sequence:

```
L      R2,SYSTOP       Get value of system parameter
SUBCOM SET,MF=(L,SUBCOMP),SYSTEM=(R2)
```

tests the value of register 2 to determine the value of the SYSTEM parameter. If register 2 contains zero, then SYSTEM=NO is set. Otherwise, SYSTEM=YES is set. The (addr,mask) form allows a bit flag in storage to be used for setting these parameters. The macro call

```
SUBCOM SET,MF=(L,SUBCOMP),SYSTEM=(FLAGS,SYSFLAG)
```

will cause the bit in the byte at address FLAGS (at the offset determined by SYSFLAG) to be tested (using the instruction TM FLAGS,SYSFLAG). If this bit is zero, then SYSTEM=NO is used, otherwise SYSTEM=YES is used.

Subcommand Processor Entry Conditions

When CMS calls a subcommand processor, the register contents are:

Register	Contents
R0	Same as caller.
R1	Same as caller.
R2	Address of the SCBLOCK that contains subcommand processor information.
R12	Entry point address.
R13	24-word save area address.
R14	Return address.
R15	Entry point address.

Note: This is the standard entry point convention except that R2 points to the SCBLOCK.

Other Parameters You Can Specify on SUBCOM

The following list briefly describes some of the other parameters of the SUBCOM macro you can use (for complete details about the SUBCOM macro, see *z/VM: CMS Macros and Functions Reference*):

- UWORD — specifies an optional user word stored in the SCBWKWRD of the SCBLOCK.
- UFLAGS — specifies an optional one byte parameter in the SCBLOCK.
- INTTYPE — specifies the status of the interrupt mask.
- KEY — specifies the storage key (NUCLEUS or USER).

Defining Subcommands in Logical Saved Segments

You can also define subcommand processors by loading the module or text files in a logical saved segment. If a subcommand processor with the same name already exists, it is overridden in a stack-like manner.

Example

To create a subcommand processor named BLUE for the COLOR program, create a module and add the following record in the logical segment definition file for the MYSEG logical saved segment:

```
MODULE BLUE * (SUBCOM)  BLUE is a subcommand processor for the COLOR program
```

BLUE is entered in key zero and disabled for interrupts. The addressing mode for BLUE is determined by the way you code the program and the location in the virtual machine where the logical saved segment is loaded. To save BLUE across an abend, you must specify the SYSTEM option on the SEGMENT LOAD command or code the SYSTEM parameter on the SEGMENT LOAD macro when you load the logical saved segment. Note that subcommand processors in logical saved segments are not removed during end-of-command processing.

For more information on creating subcommand processors in logical saved segments, see *z/VM: Saved Segments Planning and Administration*.

Deleting Subcommand Processors

Use SUBCOM CLR to delete a subcommand processor from the list of SCBLOCKs that describe the current list of subcommand processor programs. For example, to clear the subcommand processor named BLUE, code

```
SUBCOM CLR,NAME='BLUE'
```

Determining if a Subcommand Processor Is Defined

Use SUBCOM QUERY to determine if a particular subcommand processor is currently defined. If it is, SUBCOM returns to register 1 the address of the SCBLOCK. If it isn't, SUBCOM stores in register 15 a return code of 1. If SUBCOM returns an address of FX'0', there are no SCBLOCKs on the SCBLOCK list (no subcommand processors are currently defined).

For example, to determine if the subcommand processor named BLUE is defined, code

```
SUBCOM QUERY,NAME='BLUE'
```

Note: The QUERY option requires a read/write parameter list, therefore, use the Execute form (MF=(E,addr)) of the macro if you require reentrant code.

Obtaining the Anchor Point of the SCBLOCK List

Use SUBCOM ANCHOR to obtain the anchor pointer for the list of SCBLOCKs that describe the current list of subcommand processor programs. The pointer to the first entry in the SUBCOM list of SCBLOCKs is returned in register 1.

For example, to obtain the anchor pointer, code

```
SUBCOM ANCHOR
```

Note: The ANCHOR option requires a read/write parameter list, therefore, use the Execute form (MF=(E,addr)) of the macro if you require reentrant code.

Immediate Commands

An **immediate command** is a CMS command that, when issued after an attention interrupt (for example, pressing ENTER), causes CMS to suspend program execution until it (CMS) processes the immediate command.

CMS has many built-in immediate commands, including HT (Halt Typing), HX (Halt eXecution), HI (Halt Interpretation), HB (Halt Batch), SO (Suspend SVCTRACE), RO (Resume SVCTRACE), and RT (Resume Typing). See the *z/VM: CMS Commands and Utilities Reference* for more information.

You can use the facilities of CMS to create your own immediate commands, for example, to override existing CMS immediate commands or to provide a way for a user to quit or query a program you create.

Creating Immediate Commands

There are four ways you can create immediate commands: (1) you can use the NUCEXT macro, (2) you can use the IMMCMD macro, (3) you can use the IMMCMD command from within an exec (see the *z/VM: CMS Commands and Utilities Reference* for details), or (4) you can load the module or text files in a logical saved segment.

Using NUCEXT to Create Immediate Commands

The IMMCMD parameter of the NUCEXT macro defines a nucleus extension as an immediate command. Using NUCEXT to define immediate commands has several advantages:

- Nucleus extensions can be created in free storage.
- Nucleus extensions can survive abend processing and end-of-command processing.
- Nucleus extensions can be invoked as exits during abend (SERVICE parameter) and end-of-command (ENDCMD parameter) processing.

Nucleus extensions established as immediate commands can be invoked explicitly (enter the command at the terminal or from an exec) or as part of normal CMSCALL or SVC 202 processing. See "Nucleus Extensions" on page 191 for details.

Using the IMMCMD Macro to Create Immediate Commands

Use IMMCMD SET to create an immediate command. If an immediate command with the same name already exists, it is overridden in a stack-like manner.

Example 1

To define an immediate command named DOIT at entry point DOITADDR, code

```
IMMCMD SET,NAME='DOIT',EXIT=DOITADDR
```

Entry Conditions

When the routine you create to process the immediate command receives control, it is disabled for interrupts and the PSW key is 0. The exit routine must not perform any I/O operations or issue any SVCs that result in I/O operations. In addition, the exit routine must not enable itself for interrupts. DIAGNOSE instructions can be used within the exit, but the exit routine must not enable itself for interruptions that may be caused by the DIAGNOSE (for example, DIAGNOSE code X'58'). On entry, the exit routine is passed the following information:

Register	Contents
R0	Address of immediate command line in extended PLIST format.

R1
Address of immediate command line in standard parameter list format. For a 31-bit mode program, register 1 contains only the address. For a program running in 24-bit mode in an XA or XC virtual machine, the high-order byte of register 1 is set to X'06' to indicate that this routine was invoked as a result of an immediate command.

R2
Address of the IMMBLOK. The IMMBLOK contains the user word and other relevant information. The format of the IMMBLOK is as follows:

Bytes	**Information**
0-3	Address of next IMMBLOK
4-7	User word
8-15	Command name
16-19	Reserved
20-23	Entry point address

R12
Entry address

R13
A thirteen doubleword save area mapped by the USERSAVE macro. The USECTYP field of USERSAVE is set to X'06' to indicate that the routine was invoked as an immediate command.

R14
Return address

R15
Entry address

Other Rules for Creating Immediate Commands

1. Immediate commands can be 1 to 8 characters in length. Synonyms can be set up for immediate commands just like they can be for regular CMS commands. Immediate commands or their synonyms must begin with a non-blank character.
2. Immediate commands are delimited by a blank. Any data following the blank is passed to the immediate command routine as parameters. The capability to pass parameters is not applicable to immediate commands declared by the IMMCMD command. Immediate commands and their parameters are subject to translation just as regular CMS commands are.
3. Immediate commands can be set up to override built-in CMS Immediate commands (for example, HX). However, built-in CMS commands cannot be cleared.
4. Immediate commands with the same name can override each other in a stack-like manner, with the most recent one declared being the one in effect.
5. The logical line-end character is ignored on immediate command input lines.

Defining Immediate Commands in Logical Saved Segments

You can also define immediate commands by loading the module or text files in a logical saved segment. When the logical saved segment containing the immediate command is loaded, the nucleus extension is activated. If an immediate command with the same name already exists, it is overridden in a stack-like manner.

Example

To define DOIT as an immediate command, add the following record in the logical segment definition file for the MYSEG logical saved segment:

```
MODULE DOIT * (IMMCMD)
```

For more information on creating immediate commands in logical saved segments, see the *z/VM: Saved Segments Planning and Administration*.

Deleting Immediate Commands

Immediate commands created by the IMMCMD macro are automatically deleted when a program returns to the CMS command environment (except when in CMS subset mode), or when CMS performs abend recovery. To explicitly delete an immediate command that was created by the IMMCMD macro, use IMMCMD CLR macro. Any previously overridden immediate command with the same name is reinstated by this action.

For example, to delete the immediate command named DOIT, code

```
IMMCMD CLR,NAME='DOIT'
```

Note: To delete an immediate command that was created by the NUCXLOAD command, the NUCEXT function, or the NUCEXT macro, use the NUCXDROP command, the NUCEXT CANCEL function, or the NUCEXT CLR macro. To delete an immediate command that was loaded as part of a logical saved segment, purge the segment or use the NUCXDROP command, the NUCEXT CANCEL function, or the NUCEXT CLR macro.

Obtaining Information about Immediate Commands

Use IMMCMD QRY to obtain information about an immediate command.A return code of 44 from QRY indicates that the immediate command does not exist.

For example, to obtain information about the immediate command named DOIT, code

```
IMMCMD QRY,NAME='DOIT'
```

Chapter 14. Abend Processing

This chapter:

- Provides an overview of CMS abend processing
- Lists the resources you can save across an abend
- Describes how to use the CMS ABNEXIT macro to create, clear, and reset abend exit routines
- Lists the OS/MVS and DOS macros you can use to create abend exit routines.

Overview of CMS Abend Processing

A program abnormally terminates (abends) when CMS detects a program interrupt or when a program issues either the OS/MVS ABEND or CMS DMSABN macro. When an abend occurs, CMS checks to see if any user exit routines have been defined for the program.

Unless an abend exit routine specifies otherwise, CMS resets or releases the resources used by the program. This includes such things as resetting pointers, releasing storage, closing files and virtual devices, and so on. (Note that your program can instruct CMS to save certain resources, such as interrupt handlers and nucleus extensions.)

Macros That Define Abend Exit Routines

There are several macros that you can use to define abend exit routines:

- ABNEXIT
- SPIE and ESPIE (OS/MVS macros)
- STAE and ESTAE (OS/MVS macros)
- STXIT PC and STXIT AB (DOS macros).

Error Event Monitors may also be defined as exit routines that get control when an AbnormalEnd CSL call is issued. The VMERROR and VMERRORCHILD events get signalled. Refer to the *z/VM: CMS Application Multitasking* manual for details.

Note that in an XC virtual machine, abend exits get control in primary space mode and if they return control to CMS abend processing, they must return it in primary space mode.

Abend Exit Routine Search Order

The search order CMS uses to check for abend exit routines depends on what caused the abend. If the:

- DMSABN macro causes the abend, CMS checks for abend exit routines defined by the VMERROR and VMERRORCHILD events. It then checks for abend exit routines defined by the ABNEXIT macro. If any exist, DMSABN passes control to the most recent one set. If none are found or if recovery is not attempted by an ABNEXIT-defined exit, CMS checks for MVS ESTAE-defined exits.
- OS/MVS ABEND macro causes the abend, CMS checks first for exit routines defined by the OS/MVS STAE or ESTAE macros. It then checks for abend exit routines defined by the VMERROR and VMERRORCHILD events. If none are found, CMS checks for exit routines defined by ABNEXIT.
- Program interrupt causes the abend, CMS checks first for exit routines defined by the OS/MVS SPIE or ESPIE macros, next for exit routines defined by the OS/MVS STAE or ESTAE macros, then for exit routines defined by the VMERROR and VMERRORCHILD events, and last for exit routines defined by ABNEXIT.
- DOS environment is active, CMS checks first for program check exit routines defined by STXIT PC. If none are found, CMS checks for a linkage to an abnormal termination routine (created by STXIT AB). If no PC or AB routine can be given control, control goes to CMS abend processing routines, which can invoke any exit routines created by ABNEXIT.

What You Can Save Across a CMS Abend

Various CMS macros include a SYSTEM parameter; this lets you save resources across an abend. For example, you can use the SYSTEM parameter of the HNDEXT macro to prevent an external interrupt handler from being released when an abend occurs. The following table lists the macros that provide the SYSTEM parameter and the resource you can save:

Table 23. Macros and the Resource the SYSTEM Parameter Saves

Macro	Resource the SYSTEM Parameter Saves
ABNEXIT	Abend exit routine
HNDEXT	External interrupt handler
HNDIO	I/O interrupt handler
HNDSVC	SVC interrupt handler
NUCEXT	Nucleus extension
SEGMENT	Saved segment
SUBCOM	Subcommand processor
SUBPOOL	Global subpool

The following table lists the CSL routine that provides the SYSTEM parameter and the resource you can save:

Routine	Resource the SYSTEM Parameter Saves
DMSSPCC	Data space

Creating Abend Exit Routines

As mentioned earlier, there are several facilities you can use to create your own abend exit routines: the OS/MVS SPIE, ESPIE, STAE, and ESTAE macros, and the CMS ABNEXIT macro. You can use these routines to try to recover from whatever caused the abend and return control to the program, perform some type of clean-up action (prior to the clean-up that CMS abend recovery routines perform), save some information that might help in debugging, or do nothing and pass control to CMS abend recovery. The ABNEXIT macro lets you create, clear, and reset abend exit routines.

Creating Abend Exits

Use ABNEXIT SET to create an abend exit routine. CMS stacks the abend routines that ABNEXIT SET creates; the most recent abend exit routine created is the first one that CMS calls.

Note: One abend exit routine cannot create another. Also, if a program check occurs while the exit routine is processing, CMS passes control to the previous exit in the list. If there are no previous exits, CMS abend recovery occurs. There are certain differences from OS/MVS in processing abend exit routines. If an abend occurs in a subtask abend exit, the parent task will get control only when it is packaged in a separate text deck.

If you want the ABNEXIT abend exit routine to survive abend recovery, you can specify the SYSTEM=YES option on the ABNEXIT macro. The abend exit routine must reside in storage that is not reclaimed during abend processing.

Entry Conditions

You must provide the proper entry and exit linkage for your abend exit routine. When your routine receives control, the register contents are as follows:

Register
Contents

R1
Address of an area of storage mapped by the CMSSDWA DSECT. To obtain the CMSSDWA expansion, call the DMSSDWA macro in the program that contains the ABNEXIT macro.

R13
Address of an 18-fullword save area (for your use).

R14
Return address.

R15
Entry point address of your exit routine.

Exit Options

When your abend exit routine completes, it can do one of two things:

1. Return control to the CMS abend recovery routines — to do this, issue a branch on register 14. CMS will call any previous abend exits if they exist; if none exist, CMS continues with normal abend recovery. Note that in an XC virtual machine, control must be returned in primary space mode.
2. Return elsewhere — to do this, load the PSW at time of abend or a modified version of the PSW. Before you load the PSW, your exit routine should issue an ABNEXIT RESET macro.

Resetting Abend Exits

If an abend exit routine returns control to a program (rather than continuing abend processing), subsequent abends bypass the exit routine unless, during its initial processing, the abend exit routine issues ABNEXIT RESET.

Note: When you use ABNEXIT RESET, you must specify the NUCON macro in the program that contains the abend exit routine.

Deleting Abend Exits

When CMS abend recovery occurs, CMS automatically clears all exit routines known to the system, except those created with the SYSTEM=YES option. Abend exits are **not** cleared at CMS end-of-command.

To explicitly clear an abend exit routine created by ABNEXIT, use ABNEXIT CLR. (Note that abend exits cannot be cleared from within an exit routine.) For example, to clear the abend exit routine at location ABEND1, code the ABNEXIT macro as follows:

```
ABNEXIT CLR,ABEND1
```

Abending a Program (DMSABN Macro)

When a program encounters an unexpected condition, it can do one of two things: it can (1) stop processing, set a return code, issue a message, and return control to the user or (2) it can issue the DMSABN macro to end the program.

When you use the DMSABN macro to force an abnormal end, you can use abend exit routines (created by the ABNEXIT macro or the MVS ESTAE macro) to determine whether you should try to recover or whether you should allow CMS abend recovery to complete. When a DMSABN-initiated abend occurs, CMS checks for abend exit routines defined by ABNEXIT and passes control to the one most recently set. If none are found or if recovery is not attempted by an ABNEXIT-defined exit, CMS checks for MVS ESTAE-defined exits. Using the DMSABN macro and ABNEXIT allow you to put less error handling code in your main program; your error handling functions can be part of a program that executes only when there is an error.

Examples

The DMSABN macro requires you to specify an abend code (0 through FFF), which appears in the DMSABE148T system termination message.

Example 1

Routines that do not reside in the nucleus should specify TYPCALL=SVC to generate CMSCALL linkage. For example, to abend a program and set an abend code of 111, a non-nucleus resident routine could code the following:

```
DMSABN  111,TYPCALL=SVC
```

or, because TYPCALL=SVC is the default value:

```
DMSABN  111
```

Example 2

Routines that reside in the nucleus or that are nucleus extensions should specify TYPCALL=BALR so that a direct branch to DMSABE is generated. For example, to abend a program and set an abend code of 222, a nucleus resident routine could code the following:

```
DMSABN  222,TYPCALL=BALR
```

Note: For DMSABN to gain addressability, you must specify the NUCON mapping macro in the program that contains the nucleus extension.

A Sample Program

The following program illustrates the use of the ABNEXIT macro.

```
TABNEX1  CSECT
         STM   R14,R12,12(R13)
         USING TABNEX1,R12
         ST    R13,SAVE+4
         LA    R13,SAVE
         XR    R3,R3
         ABNEXIT SET,EXIT=EXITSUB,UWORD=WORD,ERROR=ERROR1
         XR    R2,R2
         DR    R2,R3  IT WILL CAUSE A PROGRAM CHECK DIVIDING BY ZERO.
         XR    R2,R2
         WRTERM 'END OF MAIN ROUTINE'
OUT      L     R13,SAVE+4
         LM    R14,R12,12(R13)
         SLR   R15,R15
         BR    R14
ERROR1   EQU   *
         WRTERM ' AN ERROR OCCURED WHEN EXECUTING THE MACRO'
         B     OUT
SAVE     DS    18F
WORD     DS    F
******************EXIT ROUTINE*****************************************
         DROP  R12
EXITSUB  EQU   *
         LR    R11,R15
         USING EXITSUB,R11
         LR    R4,R1
         USING CMSSDWA,R4
         USING NUCON,0
         ABNEXIT RESET,MF=(L,LIST,SIZ)
         ABNEXIT MF=(E,LIST)
         APPLMSG  TEXT='EXIT ROUTINE ENTERED',APPLID=CMS
OUT2     L     R12,SDWREGS+8*6
         LPSW  SDWPSW
LIST     ABNEXIT MF=L
         PRINT NOGEN
         DMSSDWA
         NUCON
         REGEQU
         END
```

Part 3. Managing CMS Programs

This part of the document describes how to use CMS commands to build, load, execute, and update assembler programs and program packages. Part 3, " Managing CMS Programs," on page 209 includes the following chapters:

- Chapter 15, "Program Packaging," on page 211 provides an overview of the program packaging process.
- Chapter 16, "Assembling, Loading, and Executing Programs," on page 217 describes how to assemble your programs and load them into storage.
- Chapter 17, "Creating and Using a Callable Services Library," on page 233 describes how to create and use your own callable services library (CSL).
- Chapter 18, "Using Auxiliary Directories," on page 273 describes how to add an auxiliary directory to CMS and create an auxiliary directory.

Chapter 15. Program Packaging

This chapter:

- Lists some program packaging considerations
- Defines the different forms a program can take
- Provides an overview of the program packaging process
- Discusses how program attributes such as program life, addressing mode, residency mode, and relocatability affect your programs.

For information on how to use CMS macros to create and manage nucleus extensions, subcommands, and immediate commands, see Chapter 13, "Nucleus Extensions and Commands," on page 191.

Program Packaging Considerations

CMS provides you with the means to package your programs into several different forms, each of which has distinct characteristics. These forms include assemble files, text files, modules, nucleus extensions, subcommands, and immediate commands. Each of these forms has various inherent characteristics that you cannot change, such as how it can be executed.

In addition to these characteristics, you can assign various attributes to a package. The same program can be packaged into two different forms and have different attributes associated with it. For example, say the program, MONEY, takes advantage of bimodal CMS. When you load MONEY as a text file, you may want to assign an AMODE 24. However, because MONEY can run with 31-bit addressing, it is also possible to package it as a module and load with an AMODE of 31. Therefore, MONEY can be packaged in two different forms and each form has different attributes associated with it.

Here are some questions you need to consider for each program or program package you create:

1. How do you want to package the program? As a module? A nucleus extension? Or one of the other forms?
2. How do you want your program stored? Does it matter when the program is deleted from storage?
3. Will the program run in an XA or XC virtual machine? If so, will the program expect 24-bit addresses only, 31-bit addresses only, or a combination of both?
4. Will the program reside in the transient area, at a specific address, at the first available free storage location, or, for XA and XC programs only, above the 16MB line?
5. If the program is a module, will it be relocatable? Do you want to restrict it to an XC virtual machine or to an XA virtual machine or to a 370 virtual machine?

   Note: Only CMS levels prior to CMS Level 12 can execute in a 370 virtual machine.

Program Packaging Overview

Following is a more detailed list of the various forms a program can take.

Assemble file
: The actual assembler language program. It cannot be executed as is.

Text file
: The output of the assemble process (the process of compiling, or assembling, an assemble file). To execute it, you must load it and start it. CMS text files are similar to OBJECT files in MVS.

Module
: A program package that consists of one or more text files that have been linked. To execute a module, you can enter its name from the command line at your terminal.

Nucleus extension
A nucleus extension is a program in free storage that CMS treats as if it were a nucleus command. This means that, once loaded, the nucleus extension remains in storage; when the nucleus extension is called, CMS does not have to read it into storage from a disk. For more information, see "Nucleus Extensions" on page 191.

Subcommand
A subcommand environment is one where a program defines to CMS a number of routines, or subcommands, that CMS can then recognize and invoke by name. For more information, see "Subcommand Environments" on page 197.

Immediate command
An immediate command is a command that, when issued after an attention interrupt (for example, pressing ENTER), causes CMS to suspend program execution until it (CMS) processes the immediate command. For more information, see "Immediate Commands" on page 200.

Program Packaging — A Simple Scenario

To understand the various program packaging options available to you, it helps to understand how the program packaging process works. For example, assume you are designing and developing an assembler language program named DOITALL. From development to execution, DOITALL can be in one of three basic formats, each of which is identified by its file type:

1. An assemble file (DOITALL ASSEMBLE) — This is the file that contains the actual assembler language program. Before you can run DOITALL, you need to compile (or assemble) it, thus creating a text file. You can assemble a program using one of the following:

   a. With Assembler XF, use the CMS ASSEMBLE command.

   b. With Assembler H, use the HASM command.

   c. With the high level assembler, use the HLASM command.

   For more information on assembling files, see Chapter 16, "Assembling, Loading, and Executing Programs," on page 217.
2. A text file (DOITALL TEXT) — This is the output of the assembly process. To execute a text file, you must load it into storage and start it. There are several ways to load and start text files, see "Loading and Executing Text Files" on page 221 for details.
3. A module file (DOITALL MODULE) — Module files, or modules, consist of one or more text files that have been linked to form a single program package. After the required text files are loaded into storage, you can use the CMS GENMOD to create a module. See "Generating and Executing Modules" on page 227 for details.

In addition to the three basic forms a program can take (assemble file, text file, and module file), you can package programs as nucleus extensions, subcommands, or immediate commands. For example, you could create a nucleus extension, a subcommand, or an immediate command from either DOITALL TEXT or DOITALL MODULE.

Program Life - Determining How Long Your Program Stays in Storage

Program life refers to how long a program remains in storage. In CMS, program life is determined by the program package (text file or module file) and by the method used to load the program.

Knowing how and when the various program types are deleted from storage can help you select the program package and load method most suitable for your application.

The following list summarizes when programs are deleted according to the method used to load the program.

1. **LOAD, INCLUDE —** in general, text files loaded with the LOAD and INCLUDE commands remain in storage until you issue another LOAD or LOADMOD command or until CMS abend recovery occurs. Some further considerations follow:

   a. You can use the PRES option of the LOAD and LOADMOD commands to prevent deletion of programs previously loaded with LOAD, INCLUDE, or LOADMOD; however, if a program you load requires storage that is currently held by another program, the other program is deleted regardless of whether you specify the PRES option or not. (In prior releases, the previously loaded program would be overlaid but not deleted.)

   b. Because CMS deletes programs rather than overlaying them (see the previous note), the SET LOADAREA command setting can affect the program life of a non-relocatable program. When LOADAREA=20000, CMS loads text files at storage location X'20000' unless an ORIGIN is specified. For example, assume that (a) you have two programs, OLDPROG and NEWPROG, (b) LOADAREA=20000 is in effect, and (c) you issue the following sequence of commands:

      ```
      load oldprog
      load newprog (pres
      ```

      Because OLDPROG has no ORIGIN specified, CMS loads it atX'20000' Because NEWPROG also has no ORIGIN specified, CMS loads it at X'20000'. Because OLDPROG resides in storage that NEWPROG needs, OLDPROG is deleted, even though the PRES option was specified.

      On the other hand, when LOADAREA=RESPECT (the default value in XA and XC virtual machines), CMS loads text files at the largest contiguous area of storage available unless an ORIGIN is specified.

      Therefore, if LOADAREA=RESPECT you issue the following sequence of commands:

      ```
      load oldprog
      load newprog (pres
      ```

      OLDPROG is **not** deleted. Both OLDPROG and NEWPROG will reside in the largest contiguous pieces of storage available at the time they were loaded.

   c. If you issue an INCLUDE command with a specified ORIGIN that requires storage currently occupied by programs that were specified by the LOAD command, unpredictable results may occur when a START command is issued.

2. **LOADMOD —** modules loaded with the LOADMOD command remain in storage until you issue a subsequent LOAD or LOADMOD command, or until CMS abend processing occurs. Some further consideration follow:

   a. You can use the PRES option of the LOAD and LOADMOD commands to prevent deletion of programs previously loaded with LOAD, INCLUDE, or LOADMOD; however, if a program you load requires storage that is currently held by another program, the other program is deleted regardless of whether you specify the PRES option or not. (In prior releases, the previously loaded program would be overlaid but not deleted.)

   b. Because CMS deletes programs rather than overlaying them (see the previous note), the SET LOADAREA command setting can affect the program life of a non-relocatable program. See "1.b" on page 213 for a further discussion.

3. **Command Invocation —** modules that are invoked as commands (invoked by the file name associated with the MODULE file) remain in storage until the command completes or CMS abend recovery.

   Note: Attempting to use the START command to restart a module that has been invoked as a command can cause unpredictable results.

4. **NUCXLOAD —** modules that you use the NUCXLOAD command to invoke as nucleus extensions remain in storage until:

   a. You issue the NUCXDROP command to delete the program.

   b. CMS abend recovery (**unless** you issue the SYSTEM option of the NUCXLOAD command). If you issue the SYSTEM option of NUCXLOAD, the program is **not** deleted during CMS abend recovery.

   c. You log off your virtual machine.

5. **CMSCALL** — if you use the CMSCALL macro to invoke a module, the module remains in storage until it completes or CMS abend recovery.
6. **OS/MVS LOAD Macro** — programs invoked with the OS/MVS LOAD macro remain in storage until:

   - they complete (for example, end of command)
   - they are deleted by the OS/MVS DELETE macro
   - they are deleted to provide storage for subsequent programs you load
   - CMS abend recovery

   Some further considerations follow:

   a. If you use the OS/MVS LOAD macro to load a text file that (a) has already been loaded with the LOAD command and (b) is still identified in the CMS loader tables, CMS does **not** reload the text file; rather, it reuses the program currently loaded. The length returned in R1 under these circumstances is unpredictable.
   b. A program loaded using the OS/MVS LOAD macro may be deleted if it resides in storage required by a non-relocatable program you subsequently load.
   c. Programs loaded using the OS/MVS LOAD macro are not automatically deleted if the CMS LOAD or LOADMOD commands are subsequently issued within the same command cycle.
   d. If you use the OS/MVS LOAD macro to bring your program into storage and you issue a LINK macro for the same program, the copy of the program already in memory will be used.
   e. A program which is loaded into memory by the LINK macro will be deleted during ENDSVC processing.
   f. If you use the OS/MVS LOAD macro to load a module from CMS LOADLIB and this module was marked as nonreusable or nonreentrant, the LOAD macro will always bring in a new copy of the load module. The previous copy of the load module will be deleted.

Addressing and Residency Modes

As mentioned in "Bimodal Addressing" on page 24, programs that run in XA or XC virtual machines can use 24-bit addresses, 31-bit addresses, or a combination of both. Programs that use 31-bit addresses can run anywhere in a virtual machine's addressing space; programs that use 24-bit addresses must run below the 16MB line. While 31-bit addressing can alleviate program storage restraints, it also requires you to learn some new terminology.

Addressing mode refers to the type of address (31-bit or 24-bit) a program expects to handle when it receives control. A program's AMODE attribute determines its addressing mode:

- AMODE 24 — Means a program can handle 24-bit mode addresses only. An AMODE 24 program must reside below the 16MB line.
- AMODE 31 — Means a program can handle 31-bit mode addresses. An AMODE 31 program can reside above or below the 16MB line.
- AMODE ANY — Means you are deferring the decision to assign the program an addressing mode. There are several points in the program cycle when you can assign an AMODE or override an existing AMODE. You can also use AMODE ANY to let the program's addressing mode default to the value of the program that called it.

Residency mode refers to where a program resides when CMS loads it (above or below the 16MB line). A program's RMODE attribute determines its residency mode:

- RMODE 24 indicates that CMS loads the program below the 16MB line.
- RMODE ANY indicates that CMS loads the program above 16MB unless insufficient storage is available above the 16MB line.

When You Can Set Addressing and Residency Modes

You can define, or redefine a program's addressing and residency mode at virtually every point in the program development cycle. This helps to provide program packaging flexibility: a text file you load can have different AMODE and RMODE values than the text file created during assembly; a module you create can have different program attributes than the text file(s) from which you create it.

The following list describes the different times you can set the addressing mode and residency mode for a program:

- **At assembly time** — You can code (using Assembler H or the high level assembler) the AMODE and RMODE instructions in the assemble file to define addressing mode and residency mode values. When you assemble the program, CMS associates the AMODE and RMODE values you defined with the resulting text file.
- **At loading time** — When you load a program, you can use the AMODE, RMODE, and ORIGIN options of the LOAD command to specify addressing mode and residency mode values. AMODE and RMODE values set by the LOAD command override any values set at assembly time. (Note that the RMODE and ORIGIN option on the LOAD command are mutually exclusive.)

  For more information on loading programs, see Table 25 on page 222.
- **At module generation time** — You can use the AMODE and RMODE options of the BIND or GENMOD command to specify addressing mode and residency mode values for the module you generate. AMODE and RMODE values set by the BIND command override any values set at assemble time. The AMODE value set by the GENMOD command overrides the AMODE value set at assembly time or on the LOAD command. For relocatable modules, the RMODE value set by the GENMOD command overrides the RMODE value set at assembly time or on the LOAD command. For non-relocatable modules, the RMODE value specified on the GENMOD command is ignored; and the module is loaded based on the location of the text file(s) when you issued the GENMOD command.

  Note: If you do not specify an RMODE on GENMOD, and the CSECTs in the module have different RMODEs, CMS assigns an RMODE of 24 to the module.

Program Attribute Default Values

If, during the program packaging process, you do not specify an addressing mode or residency mode for a program, the default values are:

Table 24. Program Attribute Default Values

Specified	Default Value
Neither	AMODE 24 RMODE 24
AMODE 24	RMODE 24
AMODE 31	RMODE 24
AMODE ANY	RMODE 24
RMODE 24	AMODE 24
RMODE ANY	AMODE 31

Notes:

1. The combination AMODE 24 RMODE ANY is invalid.
2. The CMSCALL macro calls AMODE ANY programs in the addressing mode of the caller.

Switching Addressing Mode from within a Program

Note that there may be times when part of a program needs to run in a different addressing mode than the rest of the program. To switch addressing modes, you can use the AMODESW macro. For more information, see Chapter 4, “Program Invocation - Direct Branch Linkage,” on page 35.

Chapter 16. Assembling, Loading, and Executing Programs

How you assemble your programs depends on what assembler you use. If you use Assembler H, use the HASM command to assemble files. If you use Assembler XF, use the CMS ASSEMBLE command. If you are using the High Level Assembler, use the HLASM command.

Note: The assembler invoked by the CMS ASSEMBLE command does not recognize AMODE and RMODE instructions.

This chapter describes how to:

- Assemble programs into text file and load them into storage
- Create modules from text files and load them into storage
- Display information about programs in storage.

Assembling Programs

The ASSEMBLE, HASM, and HLASM commands convert assembler language source programs into text file format. When you invoke these commands, CMS searches all of your accessed disks, using the standard search order, until it locates the specified file. The output created by the assembler includes a listing and text deck. The file name of the listing and text deck is the same as the file name of the input file; the file types are LISTING and TEXT, respectively. The listing and text deck are written to disk according to the following priorities:

- If the source file is on a read/write minidisk or directory, the TEXT and LISTING files are written onto that minidisk or directory.
- If the source file is on a read-only minidisk or directory that is an extension of a read/write minidisk or directory, the TEXT and LISTING files are written onto the read/write minidisk or directory.
- If the source file is on any other read-only minidisk or directory, the TEXT and LISTING files are written onto the first read/write minidisk or directory.
- If the source file is on tape or in your virtual reader, the TEXT and LISTING files are written onto the first read/write minidisk or directory.
- If none of the above choices are available, the command is terminated.

  In addition, the assembler creates work files (SYSUT1, SYSUT2, SYSUT3) on the same file mode used for the TEXT and LISTING files. The assembler erases these files if it runs to completion, but these files may remain on your disk or directory if execution is interrupted. In either case, the assembly cannot run unless there is sufficient space available to create the SYSUTx work files.

Example 1

To assemble a source program named MYFILE ASSEMBLE on your A-disk, enter

```
hasm myfile
```

or

```
hlasm myfile
```

or, if you are using Assembler XF,

```
assemble myfile
```

CMS will create two files on your A-disk: MYFILE TEXT and MYFILE LISTING.

Example 2

To assemble MYFILE ASSEMBLE and specify that the listing file (MYFILE LISTING) be printed rather than kept on your A-disk, enter

```
hasm myfile (print
```

or

```
hlasm myfile (print
```

or, for Assembler XF, enter

```
assemble myfile (print
```

For more information on the HASM command, see the *Assembler H Version 2 Application Programming Guide*. For more information on the HLASM command, see the *High Level Assembler/MVS & VM & VSE Programmer's Guide*. For more information on the ASSEMBLE command, see *z/VM: CMS Commands and Utilities Reference*.

Identifying Files

When you use the ASSEMBLE, HASM, or HLASM command, CMS automatically issues several FILEDEF commands to set up default file definitions. These file definitions describe the input and output files used by the assembler. The input and output files used by the assembler are:

ddname	**File Description**
ASSEMBLE	ddname SYSIN — source; input to the assembler
CMSLIB	ddname SYSLIB — macro libraries; input to the assembler
TEXT	ddname SYSLIN — object; output of the assembler
LISTING	ddname SYSPRINT — assembly listing; output of the assembler
PUNCH	ddname SYSPUNCH — output of the assembler
SYSUT1	SYSUT1 — assembler work file
SYSUT2	SYSUT2 — assembler work file
SYSUT3	SYSUT3 — assembler work file

Default File Definitions

The default FILEDEF commands issued by the CMS ASSEMBLE command for these input and output files are:

```
FILEDEF ASSEMBLE DISK fn ASSEMBLE fm (RECFM FB LRECL 80 BLOCK 800
FILEDEF TEXT DISK fn TEXT fm
FILEDEF LISTING DISK fn LISTING fm (RECFM FBA BLOCK 1210
FILEDEF PUNCH PUNCH
FILEDEF CMSLIB DISK DMSGPI MACLIB * (RECFM FB LRECL 80 BLOCK 800
FILEDEF SYSUT1 DISK fn SYSUT1 fm4 (BLOCK 13030 AUXPROC asmproc
FILEDEF SYSUT2 DISK fn SYSUT2 fm4 (BLOCK 13030 AUXPROC asmproc
FILEDEF SYSUT3 DISK fn SYSUT3 fm4 (BLOCK 13030 AUXPROC asmproc
FILEDEF SYSTERM TERMINAL (AUXPROC termproc
```

Determining What File Definitions Are in Effect

To find out what file definitions are currently in effect, enter

```
filedef
```

or

```
query filedef
```

Overriding Default File Definitions — Examples

If you issue a FILEDEF command using one of the assembler ddnames before you issue the ASSEMBLE, HASM, or HLASM commands, it will override the default file definitions. At completion of the ASSEMBLE, HASM, or HLASM command, all FILEDEFs that do not have the PERM option are erased.

1. The default ddname for the source input file (SYSIN) is ASSEMBLE. To instruct the assembler to read your input file from your card reader and assign the file name SAMPLE to the output TEXT and LISTING files, enter:

   ```
   filedef assemble reader
   hasm sample
      or
   filedef assemble reader
   hlasm sample
   ```

   or, for Assembler XF, enter:

   ```
   filedef assemble reader
   assemble sample
   ```

2. To assemble a source file named OS.SOURCE.FILE directly from an OS/MVS disk and assign it the name MYFILE ASSEMBLE, enter:

   ```
   filedef assemble disk myfile assemble b4 dsn os source file
   hasm myfile
      or
   filedef assemble disk myfile assemble b4 dsn os source file
   hlasm myfile
   ```

   or, for Assembler XF, enter:

   ```
   filedef assemble disk myfile assemble b4 dsn os source file
   assemble myfile
   ```

3. LISTING and TEXT are the ddnames assigned to the SYSPRINT and SYSLIN output of the assembler. To assemble MYFILE ASSEMBLE and instruct CMS to name the LISTING and TEXT files ASSEMBLE LISTFILE and ASSEMBLE TEXTFILE (rather than MYFILE LISTING and MYFILE TEXT), enter:

   ```
   filedef listing disk assemble listfile a
   filedef text disk assemble textfile a
   hasm myfile
      or
   filedef listing disk assemble listfile a
   filedef text disk assemble textfile a
   hlasm myfile
   ```

   or, for Assembler XF, enter:

   ```
   filedef listing disk assemble listfile a
   filedef text disk assemble textfile a
   assemble myfile
   ```

Identifying Libraries

When assembling your source program, you may want to include calls to macros, text files, or CSL routines that reside in CMS libraries. To do this, you must identify the library or libraries containing the code (using the GLOBAL command) before you invoke the assembler. Otherwise, the assembler will not find the code and you will receive an assembly error.

To identify the libraries to be searched, use the GLOBAL command. For example, if you have a program that uses the SEGMENT macro and some of the simulated OS/MVS macros, you would have to issue following the command:

```
global maclib dmsgpi osmacro
```

The libraries you specify on a GLOBAL command line are searched in the order you specify them. So in the previous example, DMSGPI MACLIB would be searched first and OSMACRO MACLIB second. The assembler will find the SEGMENT macro in DMSGPI MACLIB and the MVS macros in OSMACRO MACLIB.

A GLOBAL command remains in effect for the remainder of your terminal session, until you issue another GLOBAL MACLIB command or IPL CMS again. To find out what macro libraries are currently available for searching, issue the command:

```
query maclib
```

You can reset the libraries or the search order by reissuing the GLOBAL command. To clear the global list, issue the GLOBAL command with no file names:

```
global maclib
```

Macro Libraries

You can create your own macro libraries or use the macro libraries on the CMS system disk. (For more information on creating your own macro libraries, see Chapter 17, "Creating and Using a Callable Services Library," on page 233.) The macro libraries that are on the system disk contain CMS, OS/MVS, and CP assembler language macros that you may want to use in your programs.

These MACLIBs are:

- DMSGPI contains the CMS programming interface macros. In prior releases, these macros were in DMSSP MACLIB and CMSLIB MACLIB, which no longer exist.
- DMSOM contains mostly CMS internal macros. The TEOVEXIT, IO, CMSCB, and DMSJNEPL macros are the only macros in DMSOM that you can use as a programming interface.
- OSMACRO contains the macros that CMS provides for execution of programs using MVS interfaces in 370, XA, or XC virtual machines.
- MVSXA contains the simulated MVS/XA versions of the OS/MVS macros for the execution of programs using MVS interfaces in XA or XC virtual machines.
- OSMACRO1 contains the non-simulated versions of OS/MVS macros that are used only for assembly on CMS.
- OSVSAM contains the subset of OS/VSAM macros which CMS supports.
- HCPGPI contains CP programming interface macros.
- HCPPSI contains CP programming interface macros.

To obtain a list of macros in any of these libraries, use either the MACLIST command or the MACLIB command with the MAP function. In the MACLIST environment, you can issue CMS commands against the members directly from the displayed list. For more information on these commands, see the *z/VM: CMS Commands and Utilities Reference*.

Note: You should not use OS/MVS macros to create or update CMS macro libraries. OS/MVS partitioned data sets and CMS MACLIBs have different formats and use certain common fields differently. For example, OS defines directory size in blocks and CMS MACLIB defines directory size in bytes.

CSL Routines

Failure to load VMLIB can cause unpredictable results. The sample system profile (SYSPROF EXEC) loads VMLIB. If the call to RTNLOAD has been removed from SYSPROF EXEC, you can still have VMLIB loaded automatically by adding this line to your PROFILE EXEC:

```
RTNLOAD * (FROM VMLIB SYSTEM GROUP VMLIB)
```

VMMTLIB is automatically loaded before the system profile. For more information on the routines in VMMTLIB, see *z/VM: CMS Application Multitasking*.

You may also create your own library of CSL routines and make it available to programs. For more information on how do to this, see Chapter 17, “Creating and Using a Callable Services Library,” on page 233.

Text Libraries and Load Libraries

A text library (TXTLIB file type) contains files with a file type of TEXT. These TEXT file are relocatable object modules that are created after you compile your program. TXTLIBs are referenced when you use the CMS LOAD or INCLUDE command to create nonrelocatable modules. Also, certain TXTLIBs are referenced at run time.

Load libraries (file type LOADLIB) are link-edited programs that make use of certain OS macros such as LINK, LOAD, ATTACH, and XCTL. These macros require special handling by CMS at execution time, which is provided by the OSRUN command.

You can create your own TXTLIBs and LOADLIBs and make them available to programs at execution time. For more information, see the *z/VM: CMS Application Development Guide*.

Loading and Executing Text Files

The assembly process described in the previous sections creates a text file. To execute the text file, you need to load it and start it. To load a text file, you can use the LOAD command. To load subsequent text files, you can use the INCLUDE command. To start a text file, you can use the START command or the START option of the LOAD command. (If you are unfamiliar with loading text files, make sure to read “Program Life - Determining How Long Your Program Stays in Storage” on page 212 and “Addressing and Residency Modes” on page 214.)

The INCLUDE command has much the same format and option list as the LOAD command. The main difference is that when you issue the INCLUDE command the loader tables are not reset. If you issue two LOAD commands in succession, the second LOAD command replaces the first.

Conversely, the INCLUDE command, which you must issue when you want to load additional files into storage, should not be used unless you have just issued a LOAD command. You may specify as many INCLUDE commands as necessary following a LOAD command to load files into storage.

When the LOAD and INCLUDE commands execute they produce a load map indicating the entry points loaded, their virtual storage locations, and their AMODE and RMODE values. You may find this load map useful in debugging your programs.

The load map is written onto your A-disk as LOAD MAP A5. Each time you issue the LOAD command, the old load map is replaced by a new one. However, if you specify the NOMAP option, the old LOAD MAP file is erased and a new LOAD MAP file is not created.

File Loading Techniques

To load a text file, you can:

1. Issue the LOAD command (or a combination of LOAD and INCLUDE commands) from your terminal, from an exec, or with the CMSCALL macro from a program.
2. Issue the OS/MVS LOAD macro from a program.

Loading Text Files into Storage

Loading a text file into storage is simple; you issue the LOAD command specifying the name of a text file you want to load. Determining where the text file gets loaded can be somewhat more complex. Where CMS loads programs depends on a number of factors, including:

1. The mode of the virtual machine.
2. Whether the file is transient (as specified by the ORIGIN TRANS option of the LOAD command).
3. The setting of the SET LOADAREA command. In a 370 virtual machine, the default value of the SET LOADAREA command is LOADAREA=20000. In an XA or XC virtual machine, the default value of the SET LOADAREA command is LOADAREA=RESPECT.
4. The residency mode of the program (in XA and XC virtual machines only).

Table 25 on page 222 summarizes how the various options determine where a program is loaded.

Table 25. Where CMS Loads Programs

LOAD Command	**SET LOAD- AREA Setting**	**370 Mode Result**	**XC or XA Mode Result**
LOAD pgma ...	20000	CMS loads pgma at X'20000'	CMS loads pgma at X'20000' This overrides the RMODE value (if any) set in the text file.
LOAD pgma (RMODE 24	20000	CMS loads pgma at X'20000'	CMS loads pgma at X'20000' This overrides the RMODE value (if any) set in the text file.
LOAD pgma (RMODE ANY	20000	CMS loads pgma at X'20000.'	CMS loads pgma at X'20000' This overrides the RMODE value (if any) set in the text file.
LOAD pgma (ORIGIN TRANS	20000	CMS loads pgma at the start of the transient area.	CMS loads pgma at the start of the transient area.
LOAD pgma (ORIGIN hexloc	20000	CMS loads pgma at hexloc.	CMS loads pgma at hexloc.

Table 25. Where CMS Loads Programs (continued)

LOAD Command	SET LOAD- AREA Setting	370 Mode Result	XC or XA Mode Result
LOAD pgma ...	RESPECT	CMS loads pgma at the largest available contiguous free storage area.	CMS loads pgma at the largest available contiguous free storage area according to the first RMODE setting in the text file. If RMODE is omitted or if the first RMODE setting in the text file is RMODE 24, CMS loads the program in the largest area below 16MB. If the first RMODE setting in the text file is RMODE ANY, CMS loads the program in the largest area above 16MB. (If CMS encounters a subsequent setting of RMODE 24 in the text file, it stops loading the program above 16MB, issues a message, and starts loading the program below 16MB.)
LOAD pgma (RMODE 24	RESPECT	CMS loads pgma at the largest available contiguous free storage area.	CMS loads pgma at largest contiguous free storage area under 16MB.
LOAD pgma (RMODE ANY	RESPECT	CMS loads pgma at the largest available contiguous free storage area.	CMS loads pgma at largest contiguous free storage area above 16MB (if available).
LOAD pgma (ORIGIN TRANS	RESPECT	CMS loads pgma at the start of the transient area.	CMS loads pgma at the start of the transient area.
LOAD pgma (ORIGIN hexloc	RESPECT	CMS loads pgma at hexloc.	CMS loads pgma at hexloc.

Note:

1. The combination AMODE 24/RMODE ANY is invalid. Specifying AMODE 24 and an ORIGIN address greater than 16MB is also invalid.
2. In a 370 virtual machine, RMODE and AMODE have no affect on how CMS loads the program; however, CMS does pass the values on to the GENMOD process. This lets you develop programs on a 370 virtual machine for execution on an XA or XC virtual machine.

   Only CMS levels prior to CMS Level 12 will execute in a 370 virtual machine.
3. Using the INCLUDE command:
   a. If, after you load a program above 16MB, CMS encounters an RMODE 24 in a text file you INCLUDE, CMS restarts the load process below 16MB. Note that CMS restarts the load in the existing environment—if you change your virtual machine environment (for example, release disks) between the time you LOAD a file and the time you INCLUDE a file, unpredictable results may occur.

b. If you specify an ORIGIN address on the INCLUDE command, that address must be on the same side of the 16MB line as the program already loaded; otherwise, the INCLUDE command fails. For example, if you LOAD a program named PIE above 16MB and you attempt to INCLUDE a program named ASLICE at an ORIGIN below 16MB, the INCLUDE command fails and ASLICE does not get loaded; when PIE runs it will be missing ASLICE.
c. If you (a) specify an ORIGIN address on the INCLUDE command that requires storage currently occupied by programs loaded with the LOAD command, and (b) issue the START command, unpredictable results may occur.

Other LOAD and INCLUDE Options

In addition to the attributes listed above, there are numerous other functions of the LOAD and INCLUDE commands you might consider using. These functions include:

- Changing the entry point where control is passed when execution begins (RESET option).
- Controlling how CMS resolves references and handles duplicate CSECT names (AUTO, LIBE, and DUP options).
- Clearing storage to binary zeros before loading files (CLEAR option).
- Saving history information from the text files (HIST and NCHIST options). The HIST option saves history information (comments). The NCHIST options saves only non-commented history information. If neither the HIST nor NCHIST is specified on the LOAD or INCLUDE commands, the history information is not saved for the files being loaded into storage.
- Controlling how the CMS Loader handles the high-order bit in resolving the VCON addresses (HOBSET, NOHOBSET, and HOBSETSD).

For more information, see *z/VM: CMS Commands and Utilities Reference*.

Executing TEXT Files

After you load a text file into storage, you can execute it.

Example — Loading and Starting a Program

To load and start the source program ODDJOB ASSEMBLE, enter the following sequence of commands:

```
load oddjob (options…
start
```

or

```
load oddjob (start options…
```

Using the START option of the LOAD command passes control to the first entry point in your program. The *options…* refer to the AMODE, RMODE, ORIGIN, NORLDSAV, NOPRES and other options available on the LOAD command.

Example — Starting a Program at a Specific Entry Point

To begin execution at an entry point other than the first, you can specify the alternate entry point or CSECT name on the START command. For example, if ODDJOB has several entry points and you want to start execution at the one named CLEANUP, enter

```
start cleanup
```

Or, you can start execution at a different entry point by specifying the entry point on the LOAD (or INCLUDE) command with the RESET option.

```
load oddjob (reset cleanup
start
```

Example — Passing a Parameter List on the START Command

If the program you are going to execute expects a parameter list, you can specify the arguments on the START command line. If the user arguments are specified, the entry or * operands must be specified; otherwise, the first argument is taken as the entry point. Arguments are passed to the program in register 1. The entry operand and any arguments become a string of doublewords, one argument per doubleword, and the address of the list is placed in general register 1.

For example, to pass the argument 007 to the default entry point (the first CSECT) in the ODDJOB program, enter

```
start * 007
```

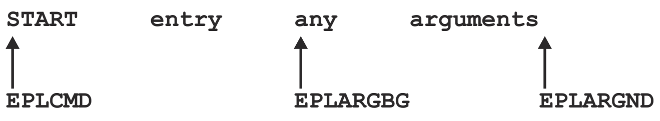
to pass the argument JIMBOND to the entry point CLEANUP in the ODDJOB program, enter

```
start cleanup jimbond
```

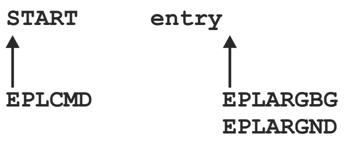
If START is issued from the virtual console or from an EXEC 2 EXEC, register 0 points to an extended parameter list block. The extended parameter list for the START command pointed to by register 0 has the following structure:

```
DC     A(EPLCMD)
DC     A(EPLARGBG)
DC     A(EPLARGND)
DC     A(0)
```

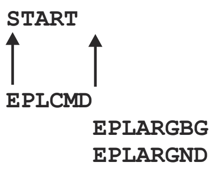
where:



or:



or:



Resolving External References

When you issue the LOAD or INCLUDE commands to load files into storage, the loader checks for unresolved references. If there are any, the loader searches your disks for text files with file names that match the external entry name. When it finds a match, the loader loads the text file into storage. If it does not find a match, the loader searches any available TXTLIBs for members that match. If there are still unresolved references, for example, if you load a program that calls routines PRINT and ANALYZE but the loader cannot locate them, you receive the message:

```
The following names are undefined:
 PRINT
 ANALYZE
```

You can issue the INCLUDE command to load additional text files or TXTLIB members into storage so the loader can resolve any remaining references. For example, if you did not identify the TXTLIB that contains the routines you want to call, you may enter the GLOBAL command followed by the INCLUDE command:

```
global txtlib newlib
include print analyze (start
```

A failure to resolve external references might occur if you have TEXT files with file names that are different from either the CSECT names or the entry names. You must explicitly issue LOAD and INCLUDE commands for these files.

At execution time, if there are still any unresolved references, their addresses are all set to 0 by the loader; so any attempt to address them in a program may result in a program check.

Loader Control Statements

In addition to the options provided with the LOAD and INCLUDE commands, you can use loader control statements to control the execution of TEXT files. These can be inserted in TEXT files, using the CMS editor.

The loader control statements allow you to:

- Set the location counter to specify the address where the next TEXT file is to be loaded (SLC* statement).
- Modify instructions and constants in a TEXT file, and change the length of the TEXT file to accommodate modifications (Replace and Include Control Section statements).
- Change the entry point (ENTRY statement).
- Nullify an external reference so that it does not receive control when it is called, and you do not receive an error message when it is encountered (LIBRARY statement).

Determining Program Entry Points

When you load a single TEXT file or a TXTLIB member into storage for execution, the default entry point is the first CSECT name in the object file loaded. You can specify an alternate entry point on the LOAD, INCLUDE, or START commands.

When you load multiple TEXT files (either explicitly or implicitly by allowing the loader to resolve external references), you also have the option of specifying the entry point on the LOAD, INCLUDE, or START command lines.

If you do not specifically name an entry point, the loader determines the entry point for you according to the following hierarchy:

1. An entry point specified on the START command
2. The last entry specified with the RESET option on a LOAD or INCLUDE command
3. The name on the last ENTRY statement that was read
4. The name on the last LDT statement that contained an entry name that was read
5. The name on the first assembler- or compiler-produced END statement that was read
6. The first byte of the first control section loaded.

For example, if you load a series of TEXT files that contain no control statements and do not specify an entry point on the LOAD, INCLUDE, or START commands, execution begins with the first file that you loaded. If you want to control the execution of program subroutines, you should be aware of this hierarchy when you load programs or when you place them in TXTLIBs.

An area of particular concern is when you issue a dynamic load (with the OS/MVS LINK, LOAD, or XCTL macros) from a program, and you call members of CMS TXTLIBs. The CMS loader determines the entry point of the called program and returns the entry point to your program. If a TXTLIB member that you load has a VCON to another TXTLIB member, the LDT card from the second member may be the last LDT card read by the loader. If this LDT card specifies the name of the second member, CMS may return that entry point address to your program rather than the address of the first member.

Text File Libraries (TXTLIBs)

You may want to keep your TEXT files in text libraries. These files have a file type of TXTLIB. You can create a TXTLIB from files with a file type of TEXT. Like MACLIBs, TXTLIBs have a directory and members. For more information, see the *z/VM: CMS Application Development Guide*.

Generating and Executing Modules

As mentioned earlier, a module consists of one or more text files that have been linked. One way to create a module is to load the required files into storage and then issue the CMS GENMOD command. For example, to create a module from DOITALL TEXT, you could enter:

```
load doitall (rldsave
genmod
```

A module can also be created in one step with the BIND command as follows:

```
bind doitall
```

Refer to *z/VM: Program Management Binder for CMS* for more information on using the BIND command.

In response to these commands, CMS generates a relocatable module named DOITALL MODULE for which all external references are resolved. To execute DOITALL MODULE, enter:

```
doitall
```

If DOITALL expects arguments passed to it, you can enter them following the module name. For example,

```
doitall today
```

Relocatable Modules

A relocatable module is one that CMS does not need to load at a specific storage location. CMS loads relocatable modules in a top down direction in the USER free storage area. (By contrast, CMS loads non-relocatable modules according to the location of the text file when the module was created). Defining your modules as relocatable helps eliminate the possibility that the storage your module requires is being used by another program.

For CMS to create a relocatable module, it (CMS) needs to have relocation (RLD) information for the program. By default, CMS does **not** save this relocation information for a text file when you use the LOAD command to create and load a text file into storage.

If you **do** want to save relocation information, you must specify the RLDSAV option of the LOAD command, or use the BIND command.

Example — Creating a Module That Is Relocatable

To create a relocatable module named OZ from text files named DORTHY, TINMAN, LION, and TOTOTOO, issue:

```
load dorthy tinman lion tototoo (rldsave
genmod oz
```

You can also use the BIND command as follows:

```
bind dorthy tinman lion tototoo (sname oz
```

When you execute OZ MODULE, CMS loads it at the highest available storage range large enough to contain it.

Example — Creating a Module That Is Not Relocatable

To create a non-relocatable module named NOTOZ from a text file named WITCH TEXT, issue the LOAD and GENMOD commands as follows:

```
load witch (norld
genmod notoz
```

Or, because NORLD is the default value, you could issue:

```
load witch
genmod notoz
```

When you execute NOTOZ MODULE, CMS loads it at the same storage location that WITCH TEXT occupies when you issue the GENMOD command.

Creating a Module to Run in the Transient Program Area

The CMS transient area, a two-page area of storage located at X'E000', is reserved for the execution of frequently used programs and commands. Programs that execute in the transient area run disabled for interrupts.

To generate a module to run in the transient area, use the ORIGIN TRANS option when you load the text file into storage, then issue the GENMOD command. For example,

```
load myprog (origin trans
genmod
```

The two restrictions placed on command modules executing in the transient area are:

1. They may have a maximum size of 8192 bytes (the size of the transient area).
2. They must be serially reusable; that is, if they are called by CMSCALL or SVC 202 and they are already loaded into the transient area, CMS does not reload them.

The CMS commands that execute in the transient area are identified in the *z/VM: CMS Commands and Utilities Reference*.

Specifying Addressing and Residency Modes for a Module

You can use the AMODE and RMODE options of the GENMOD command to specify the addressing and residency modes of a module. Note that the AMODE and RMODE values you specify on GENMOD override the values that were previously set. For example, to specify that ODDJOB run as an AMODE 31 RMODE ANY program, enter

```
load oddjob
genmod (amode 31 rmode any
```

Restricting a Module to a Specific Virtual Machine Mode

You can use the 370, XA, and XC options of the GENMOD command to specify that a module run only in a 370 virtual machine or in an XA or XC virtual machine. For example, to specify that ODDJOB run only in an XA virtual machine, enter

```
load oddjob
genmod (XA
```

Note:

1. You can override the 370 option of the GENMOD command by issuing the CMS SET GEN370 OFF command. This allows a module generated with the 370 option of the GENMOD command to run in an XA or XC virtual machine. See the *z/VM: CMS Commands and Utilities Reference* for more information on the SET GEN370 command.

2. Only CMS levels prior to CMS Level 12 will execute in a 370 virtual machine.

Saving History Information for Modules

You can use the HIST or NCHIST option of the LOAD and INCLUDE commands to create a module that includes history information from the text file used. The HIST option saves history information (comments). The NCHIST option saves only non-commented history information. If neither the HIST nor NCHIST option is specified on the LOAD or INCLUDE commands, the history information is not saved for the files being loaded into storage. For example:

```
load progone (hist
include progtwo (hist
genmod
```

The MODULE file created contains the comments that were in PROGONE TEXT and PROGTWO TEXT.

Loading Modules

To load a module file, you can:

- Issue the LOADMOD command from your terminal, from an exec, or with the CMSCALL macro from a program. You can use the ORIGIN option on the LOADMOD command to specify the load address; otherwise, CMS loads the MODULE where storage is available.
- Enter the name of the module from your terminal.
- Issue the NUCXLOAD command from your terminal, from an exec, or with the CMSCALL macro from a program.
- Issue the CMSCALL macro from a program.
- Issue the OS/MVS LOAD macro from a program.

For information on where a program is loaded, see Table 25 on page 222.

Loading a MODULE into a Saved Segment

You can load a MODULE file into a logical saved segment, a member of a CP segment space, or a discontiguous saved segment (DCSS). For a brief description of these types of saved segments, see Chapter 7, “Using Saved Segments,” on page 75. For information about defining and building saved segments, see *z/VM: Saved Segments Planning and Administration*.

Note: Building a saved segment requires the CP authority to perform the DEFSEG and SAVESEG operations.

Displaying Information about Programs in Storage

You can use the NUCXMAP and PROGMAP commands to display information about programs currently loaded in storage.

The NUCXMAP Command

Use the NUCXMAP command to display or return to a program stack information about currently defined nucleus extensions. The information NUCXMAP displays (by default) has the following format:

```
Name     Entry     Userword  Origin    Bytes     Amode  Attributes
ENDEXEC1 003EE370  00000000  003EE370  000002F8     24  SYSTEM         ENDCMD
HZ       0001B210  00000000  0001B210  00001508    ANY  SYSTEM SERVICE IMMCMD
BIGMOD   00FD2EF0  0000B848  00FD2EF0  01955050     31
```

Example 1

To display information about a specific nucleus extension, enter:

```
NUCXMAP BIGMOD
```

If more than one nucleus extension named BIGMOD exists, CMS displays information for each.

Example 2

To display information about a specific look-aside nucleus extension entry, for example, one named HZ, enter:

```
NUCXMAP HZ (ALL
```

CMS displays information on the HZ look-aside entry as well as for any nucleus extensions named HZ.

The PROGMAP Command

Use the PROGMAP command to obtain the name, entry point, origin, addressing mode, and relocation attributes of programs that you use LOAD, INCLUDE, or LOADMOD to load.

If you issue PROGMAP from within a program, use the STACK and operands of PROGMAP to have the return information placed in the program stack. To display information at your terminal, omit the STACK and FIFO|LIFO options.

Example 1

To display information about all programs, enter:

```
PROGMAP
```

In response, CMS displays something similar to the following:

```
  Name     Entry       Origin      Bytes       Attributes
  PROG1    02000400    02000400    0000066D    AMODE 31 RELOC
  PROG2    02000A6D    02000A6D    0000042A    AMODE 31 RELOC
  PROGN    00040000    00040000    00000338    AMODE 24 NON-RELOC
```

Example 2

To display information about all programs and nucleus extensions, enter:

```
PROGMAP (ALL
```

In response, CMS displays something similar to the following:

```
Name     Entry       Origin      Bytes       Attributes
PROG1    02000400    02000400    0000066D    Amode 31 Reloc
PROG2    02000A6D    02000A6D    0000042A    Amode 31 Reloc
PROG3    00040000    00040000    00000338    Amode 24 Non-reloc
Name     Entry     Userword  Origin    Bytes     Amode  Attributes
NUCX1    01000400  00004532  01000400  0000066D    ANY  SYSTEM SERVICE
NUCX2    01000A6D  00000000  01000A6D  0000042A     24
NUCX3    00020000  0000ABDC  00020000  00000338     31  SYSTEM SERVICE
```

Example 3

To display information for a program named PROG1, enter:

```
PROGMAP PROG1
```

Example 4

To display information about all nucleus extensions, enter:

```
PROGMAP (NUCX
```

Example 5

To display information about a nucleus extension named NUCX1, enter:

```
PROGMAP NUCX1 (NUCX
```

Chapter 17. Creating and Using a Callable Services Library

This chapter:

- Describes how to create your own callable services library (CSL).
- Describes how to define entry and exit points and obtain information about parameters when creating a CSL routine.
- Describes how to create a template file.
- Gives an example of a CSL routine written in Assembler.

For more information on how to invoke a CSL routine and how to make it available to users, see the *z/VM: CMS Application Development Guide*.

CSL Routines

An application program can access routines that reside in a callable services library (CSL). It is convenient for a program to call routines stored in a CSL because the calls are not resolved until the call is made (as opposed to when the module is built). This lets you make changes to a CSL routine without having to relink the routine to the application program, recompile the program, or modify any of the program's source statements. People using the application won’t be aware of the CSL.

You can invoke CSL routines from programs written in the following programming languages:

- Ada
- Assembler
- C
- COBOL[11]
- VS FORTRAN
- VS Pascal
- PL/I
- REXX.

z/VM comes with two callable services libraries, VMLIB and VMMTLIB. VMLIB contains routines that:

- Call file pool and minidisk file I/O functions
- Call file pool administration functions
- Access the current generation of REXX variables
- Issue z/VM commands through a REXX exec
- Call the CMS Extract/Replace facility, which enables application programs to obtain or modify selected system information without release or VM system dependencies
- Manipulate the CMS program stack
- Use CMS's Coordinated Resource Recovery (CRR) facility to maintain data integrity
- Use VM data spaces
- Call program-to-program communications functions using the Systems Application Architecture (SAA) Common Programming Interface (CPI) Communications (also known as SAA communications interface)
- Call SAA resource recovery (also known as CPI resource recovery) functions

[11] This pertains to the IBM OS/VS COBOL and VS COBOL II Program Products.

- Provide CMS file pool exits

VMMTLIB contains routines that:

- Call CMS application multitasking functions
- Call OpenExtensions services
- Get the value set for the workstation display address

The VMLIB routines (except those for CPI Communications and SAA resource recovery) are described in the *z/VM: CMS Callable Services Reference*. The VMLIB routines that perform CPI Communications functions are described in the *Common Programming Interface Communications Reference (https://publibfp.dhe.ibm.com/epubs/pdf/c2643999.pdf)*. The VMLIB routines that perform SAA resource recovery functions are described in the . The VMMTLIB routines for CMS application multitasking are described in *z/VM: CMS Application Multitasking*. The VMMTLIB routines for OpenExtensions services are described in *z/VM: OpenExtensions Callable Services Reference*. The VMMTLIB routine for getting the value set for the workstation display address is described in the *z/VM: CMS Callable Services Reference*.

You may also want to make your own routines and build your own library, or even customize some of the supplied VMLIB or VMMTLIB routines. This chapter describes the steps to:

1. Write individual routines to reside in a CSL
2. Create a CSL
3. Make a CSL available for application programmers to use
4. Invoke CSL routines from application programs.

Writing CSL Routines

You can write your own assembler routines to reside in a callable services library.

Your own CSL routines can work just like the ones supplied with z/VM:

1. An application program calls the routine and passes it some parameter information
2. The CSL routine does some processing using these parameters
3. The CSL routine returns control to the calling application program, passing information back in parameters.

This section describes how you can use some special macros when writing your own CSL routines, lists rules you must follow when writing your own CSL routines, and explains how to make template files that contain parameter information.

Using Macros When Writing CSL Routines

Your CSL routine must start by defining an entry point (using the CSLENTRY macro), and it must finish by defining an exit point (using the CSLEXIT macro). In the body of the CSL routine, between the entry and exit points, a CSL routine can access information about the parameters passed by the calling application program (using the CSLGETP macro). We will look at these three macros in the following sections. See the *z/VM: CMS Macros and Functions Reference* for detailed information about these macros.

Defining an Entry Point—CSLENTRY

You must specify the CSLENTRY macro before any executable code or data. Following are some examples of CSLENTRY macro calls:

```
1 CALC CSLENTRY
                       (RETURN,RESULT,OP1,OP2,DIVISOR)
2 CALC CSLENTRY DIRECT,
                       (RETURN,RESULT,OP1,OP2,DIVISOR)
3 CALC CSLENTRY DIRECT(4,1),
                       (RETURN,RESULT,OP1,OP2,DIVISOR)
4 CALC CSLENTRY OPENVM(NOCNT),
                       (RETURN,RESULT,OP1,OP2,DIVISOR)
```

In each example the CALC label generates a CSECT for the module. Each CALC CSL routine above expects a maximum of 5 parameters when it is called.

Each example above does the following:

- Saves the registers of the calling program
- Puts the address of the parameter list in register 1
- Generates the USING statement for register 15

The number of parameters, however, is dealt with in a different manner for each example. CSLENTRY does the following for each example:

Example 1
Puts the number of parameters in register 0.

Example 2
Puts the number of parameters in register 0 as well.

Example 3
Checks the number of parameters passed with the number of required and optional parameters specified on the macro. In this example there are 4 required parameters and 1 optional parameter (DIVISOR). If less than 4 parameters are passed then CSLENTRY will return a return code of -11 to the caller. If more parameters than the 5 parameters defined are passed then CSLENTRY will return a return code of -10 to the caller. If the number of parameters passed is acceptable (either 4 or 5 in this example) then the number will be stored in register 0.

Example 4
CSLENTRY ignores the number of parameters passed. Register 0 will not contain the number of parameters.

You must define a name for each parameter you expect to be passed to the CSL routine. The special parameter names RETURN and REASON should be used for a return code and reason code. In a routine that uses the OPENVM option on CSLENTRY, the special parameter name VALUE should be used if a return value is included. (In the previous examples, only RETURN is used.)

Getting Information about Passed Parameters—CSLGETP

You must use the CSLGETP macro to get information about the parameters passed by the calling application program. Following are examples of CSLGETP calls that can be used along with the previous CSLENTRY call.

```
CSLGETP   PLIST=(R2),PARM=RESULT,ADDRESS=(R6)
CSLGETP   PLIST=(R2),PARM=OP1,ADDRESS=(R5)
CSLGETP   PLIST=(R2),PARM=OP2,ADDRESS=(R6)
CSLGETP   PLIST=(R2),PARM=DIVISOR,ADDRESS=(R6)
```

In these CSLGETP examples, the address of the parameters passed to the CALC CSL routine is stored in register 2. Notice in "CALC ASSEMBLE" on page 264 that originally the address of the parameter list is stored in register 1. Because the CSLGETP macro overwrites the contents of register 1, the parameter list address must be stored in another register before calling CSLGETP. In this case we stored the plist address in register 2.

The parameters specified with PARM= correspond to the parameters specified in the CSLENTRY call. You can store the address of each parameter using ADDRESS=. For example, the address of the DIVISOR parameter is stored in register 6. The CSLGETP macro has two other parameters not discussed here. They let you obtain the length and data type of a particular parameter.

Defining an Exit Point—CSLEXIT

You must use the CSLEXIT macro at the end of the CSL routine. An example of a CSLEXIT call is:

```
CSLEXIT  RETURN=(R9),REASON=REASCODE,VALUE=RETVAL
```

CSLEXIT does the following:

- Restores the calling program's registers from the caller's save area.
- Passes a return code back to your calling program in register 15 and in the RETURN parameter.
- Passes a reason code back to the calling program in register 0 (if you specify the special parameter name REASON on CSLENTRY, then CSLEXIT also passes the reason code back to the calling program in that parameter). To get a reason code, you must also request a return code.
- Passes a return value back to the calling program in the first four bytes of the parameter that corresponds to VALUE. (The VALUE= parameter on CSLEXIT can be used only in a routine that specifies the OPENVM option on the CSLENTRY macro.)
- Returns to the calling program's address originally passed in register 14.

Rules for Coding CSL Routines

When coding CSL routines:

- Code CSL routines in assembler language, and make them re-entrant.
- Start each CSL routine with the CSLENTRY macro and complete it with the CSLEXIT macro.
- Limit routine names a maximum of 8 characters.
- Design and code CSL routines so that they can be called in either 24-bit or 31-bit addressing mode.
- Ensure that CSL routines that can be called in an XC virtual machine return control in primary space mode.
- Either specify OPENVM on the CSLENTRY macro or write the CSL routine so that the first parameter it expects is a return code parameter.
- If you want your CSL routine to be callable from programs written in various programming languages, do not include code that is applicable to just one particular programming language.
- When writing a CSL routine that will use the direct plist format, the following applies:
  - The TYPE and LENGTH operands of the CSLGETP macro cannot be used.
  - CSLGETP should be used to access the parameter addresses.
  - The parameter list size must be checked by the CSL routine. This can be done automatically by the CSLENTRY macro. CSLENTRY will return a -10 or -11 as a return code if the `req,opt` option is specified and `req+opt` =< number of parameters passed =<`req` The number of parameters passed to the routine is computed and saved for the CSL routine unless the NOCNT option is specified.
  - The plist size check done by DMSCSL against the template will, of course, not be performed when the 'DIRECT' keyword appears on the first line of the routine's template file.

See “CSL Summary and Example” on page 264 for examples on coding CSL routines.

Types of Data Supported

The data types you can use for parameters on your CSL routines are:

- Binary integer (signed or unsigned)
- Character string (fixed or variable length)
- Bit string (fixed or variable length).

You can organize these data types in various ways for use as parameters on your routines:

- Single value parameters (scalars)
- Multiple value parameters (vectors or one-dimensional arrays)
- Tabular information (two dimensional arrays: multiple columns and rows with different data types in different columns).

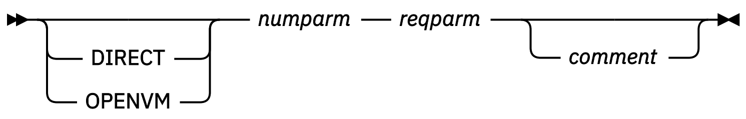
In addition, you can use a level of indirection (for example, a pointer to the data rather than the data itself).

Creating Template Files

Each CSL routine must have a template file that contains information about its parameters. This template file describes how many parameters the routine expects, what data type each parameter must be, how long each parameter must be, and whether the parameter is for input or output. A template file is specified in a control file that is used when a CSL is built. (See "Creating CSL Control Files" on page 246 for more information.)

You can create a template file using XEDIT. You can name this template file anything you want, although you may want to use a convention, such as making the file type TEMPLATE. Each parameter definition in the template file is referred to here as a template.

The first line of the template file must be in the following format:



where:

DIRECT
: indicates that the DIRECT option was specified on the CSLENTRY macro. If specified, a 1 will be displayed under the Interface attribute displayed by CSLMAP and CSLLIST. The information is available for use by any program building plists using template information. The fastpath interface is one example of this.

OPENVM
: indicates that the OPENVM option was specified on the CSLENTRY macro. If specified, a 2 will be displayed under the Interface attribute displayed by CSLMAP and CSLLIST.

  Using OPENVM specifies that the return code parameter is not required and may appear anywhere in the parameter list. The positions of the return code, reason code, and return value in the parameter list are marked using special data types.

  Specifying this plist format implies:

  - The routine is directly callable.
  - The DMSCSL interface cannot be used to call this routine.
  - Only the ADDRESS OPENVM interface can be used to call this routine from REXX.
  - The first parameter in the parameter list is not assumed to be the return code unless it is marked as such with the RTNC data type.
  - The RTNV, RTNC, and RTNR data types mark the positions in the parameter list of the return value, return code, and return reason (reason code) parameters.
  - The parameter list may contain no return code, return value, or reason code.
  - Direct calls to routines that have no return value or return code parameter will result in an ABEND if a CSL interface error is encountered.

numparm
: specifies the total number of parameters described in the file.

reqparm
: specifies the number of required parameters

comment
: is optional commentary information.

The remaining lines of the template file define the parameters, no more than one per line. These lines must be in the following format:

```
data_type    data_length    data_direction    [comment]
```

where:

data_type
specifies the template data type of the parameter being defined. The following sections detail how to describe in the template file the various data types and organizations to be used with your CSL routines. Valid template data types are:

SBIN
specifies that the parameter is a signed binary integer.

UBIN
specifies that the parameter is an unsigned binary integer.

FCHR
specifies that the parameter is a fixed-length character string.

CHAR
specifies that the parameter is a character string.

BIT
specifies that the parameter is bit data.

PTR
specifies that the parameter is a pointer to the data described by the next template entry.

TABLE
specifies that the parameters defined by this template describe a table with a fixed number of columns and a fixed or variable number of rows.

LEN
specifies the parameter that contains the length of the preceding parameter.

RTNV
specifies that the parameter contains a return value for an OPENVM routine.

RTNC
specifies that the parameter contains a return code for an OPENVM routine.

RTNR
specifies that the parameter contains a return reason (reason code) for an OPENVM routine.

These data types are discussed in detail in the following sections.

data_length
is the length of the parameter data. Valid values are:

n
specifies a numeric value designating the length of the parameter. Zero is valid only for the FCHR type and only for compatibility with previous releases of VM. A zero length specification for an FCHR parameter means that the parameter's length must be specified.

*****
specifies that the parameter's length may vary from call to call and that the parameter described by the following LEN template will contain the length at call time.

NULL
specifies that the parameter is a variable-length string terminated by a null character (X'00'). This length may be used only with the CHAR data type.

data_direction
defines the direction that the value contained in the variable is to be passed. Valid directions are:

INPUT
indicates that the parameter is used only as input to the CSL routine. In other words, the value is passed from the caller to the called routine.

OUTPUT
indicates that the parameter is used only as output from the CSL routine. In other words, the value is returned from the called routine to the caller.

INOUT
indicates that the parameter is used as both input to and output from the CSL routine. One value is passed from the caller to the called routine and another is returned from the called routine to the caller.

Note that using the same parameter for both input and output is not recommended except in the case where part of a parameter is input data and part is output on the same call. An example of this use would be a bit string where some bit values are input values and different ones are set as outputs.

Using this specification to declare a parameter that is an input parameter on some calls and an output parameter on others leads to REXX usability problems, because the caller must initialize the variable to a proper length before issuing the call even if it is an output parameter for that particular call.

Using this specification to declare a parameter that will pass an input value that will be modified by the called routine and returned as an output value is also not recommended. Use two parameters instead. The caller can specify them both as the same variable to have the input value replaced by the output value, and this avoids the need to re-initialize the value before each call.

*****
indicates that no checking of column direction is to be done for a table. This specification is valid only for the TABLE template parameter type.

comment
is optional commentary information.

Note:

1. When OPENVM is *not* specified on the first line in the template file, the first parameter template should define a 4-byte return code, for example:

```
     SBIN   4   OUTPUT
```

2. The routine name parameter, although passed as the first parameter of the CSL call, is not listed in the template file. This is because it is not passed as a parameter to the CSL routine.
3. Optional parameters must go at the end of the template file.
4. The same template file can be used for multiple routines.
5. A zero length specification for an FCHR parameter must be specified differently depending on the method used for calling the CSL routine.

   When the CSL routine is called using CALL DMSCSL, the length must be supplied as the parameter immediately following the FCHR parameter. This extra parameter for the length is not described in the template file, but is expected to be a signed binary number of four bytes length.

   When the CSL routine is called using the CSLFPI macro, the length is not specified with a separate parameter. The length is specified together with the parameter name in the"PARMS=" keyword list.

   Regardless of the method used for calling the CSL routine, the CSL will get the length using the "LENGTH=" keyword of the CSLGETP macro.

Here is an example template file for a CSL routine that gets a message from a calling program, then passes back a response and response length to the calling program. This routine has four parameters in addition to the required return code.

```
     5 5                    5 Templates defined,
                            5 parameters required
     SBIN    4   OUTPUT    Return code
     CHAR    *   INPUT     Incoming message
     LEN     4   INPUT     Length of incoming message
     CHAR   80   OUTPUT    Buffer for response
     LEN     4   OUTPUT    Length of response
```

See "CSL Summary and Example" on page 264 for other examples of template files.

Defining Parameters with Scalar Data Types

This section describes how to define the supported scalar data types using the template data types.

Note that REXX supports data only as character strings, requiring an intermediate conversion step between the call from REXX and the CSL routine. So that you can properly document the call parameters when your CSL routine is called from REXX, the format for each data type when called from REXX is also documented.

Binary Integer Parameters

Binary parameters can be either signed or unsigned. To declare a binary integer parameter, specify SBIN (for signed) or UBIN (for unsigned) as the *data_type* in the template file:

```
{SBIN|UBIN}  data_length data_direction
```

Binary parameters are always fixed length (the length cannot be different for different calls). You can specify the length in the template file as 1, 2, 3, or 4 bytes.

Scientific notation (for example, 3E2 = 3 times 10 to the second power = 300) is not supported.

The data direction for binary integer parameters can be INPUT, OUTPUT, or INOUT. (Recall from the discussion "Creating Template Files" on page 237 that the INOUT specification is not recommended.)

In REXX the data type is numeric character string. The REXX CSL interface converts the numeric character string to a binary value on input and back on output.

Character String Parameters

To define a character string parameter, specify CHAR as the *data_type* in the template file (the FCHR template data type is for compatibility only):

```
CHAR    data_length     data_direction
```

A character string parameter can be the same length for all calls or it can be a different length for each call (see "Specifying Parameter Lengths" on page 241 for a discussion on how to do this).

Specify the data length of a character string parameter in bytes. You can specify the data length as any length greater than zero (limited only by programming language constraints on length). If you specify *, you also must specify a LEN type parameter (with a direction of INPUT) on the following line to define a length parameter to follow the CHAR parameter.

The data direction for a character string parameter can be INPUT, OUTPUT, or INOUT. (INOUT is supported mainly to allow conversion of older CSL routines using the FCHR specification to the new format. IBM strongly recommends that CSL routines not be designed to use this specification.)

Bit String Parameters

To define a bit string parameter, specify BIT as the *data_type* in the template file:

```
      BIT    data_length     data_direction
```

A bit string parameter can be the same length for all calls or it can be a different length for each call (see "Specifying Parameter Lengths" on page 241 for a discussion on how to do this).

Specify the data length of a bit string parameter in bits.

A bit string must start in the first (high order) bit of the first byte of the passed parameter.

The data direction for a bit string parameter can be INPUT, OUTPUT, or INOUT. INOUT can be used for a bit string parameter in which some of the bits are for input and others are for output. IBM strongly recommends that CSL routines not be designed with bit string parameters that are inputs for some call types and outputs for others or parameters in which the same bit is both an input and an output.

For REXX calls, the parameter is a character string of "0"s and "1"s, which is translated by the REXX CSL interface to or from the bit string, depending on whether it is an input or output type parameter.

Specifying Parameter Lengths

You can define character string and bit string parameters as having either a fixed length for all calls or a length that varies from call to call.

Specify the length of fixed-length parameters in the *data_length* field of the template for the parameter, for example,

```
      CHAR    20    INPUT
```

defines a parameter that is always an input character string 20 bytes long.

The length of parameters whose size can vary from one call to the next must be specified explicitly by the calling or called routine using a length parameter. The *data_length* field in the template entry for the parameter itself must be *, indicating that a LEN parameter definition follows. For example,

```
      CHAR     *    INPUT
       LEN     4    INPUT
```

defines a parameter that is an input character string with a separate integer length parameter used to pass the length of the parameter for a particular call. The length of the LEN parameter is specified in bytes. It is an unsigned binary integer just like the UBIN template parameter. The * length specification is not supported for the LEN data type.

Another example is

```
      BIT      *    OUTPUT
       LEN     4    INPUT
       LEN     4    OUTPUT
```

which defines an output bit string parameter. The first LEN parameter contains the length (in bits) of the buffer passed on the CSL routine call, and the second LEN parameter will be set by the called CSL routine with the number of bits returned in the buffer.

Only one LEN template parameter can be specified for an INPUT parameter. An OUTPUT parameter can have one or two LEN parameters. When one is specified, if the *data_length* field for the parameter itself is *, the LEN template parameter must be INPUT and it will contain the length of the variable to hold the output value. A value specified for the *data_length* field for the parameter itself indicates the length of the buffer being provided, so the single LEN template parameter must be for OUTPUT. In this case, the LEN template parameter is being used to return the size of the output value, assuming that it can be less than the full variable size.

LEN parameters can also be used to specify the number of elements in an array parameter or the number of rows in a TABLE. See “Declaring Multi-Value Data Types in Template Files” on page 241 for more information.

Declaring Multi-Value Data Types in Template Files

Multiple value parameters—one- and two-dimensional arrays—use two special constructs in their definition: the TABLE data type and the "C." prefix to a UBIN, SBIN, CHAR, BIT, LEN, or PTR template data type specification. (See “Pointer (Indirection)” on page 243 for the definition of the PTR data type.)

Arrays and Tables

An array is a single parameter that contains multiple occurrences of a single data type and length arranged contiguously in storage, for instance, a set of five 4-byte binary integers in 20 bytes of storage, or fifteen 8-byte character strings in 120 bytes of storage. A table is a two-dimensional array, in which each column is basically an array as defined in this section.

A table is declared using two or more entries in the template file: one to describe the table and one or more to describe the columns. Each column consists of data of a single type and length, but the type and length may be different for different columns. For instance, a table with six rows and two columns could be used to pass employee names and employee numbers for six employees, where the names are 20-byte character strings and the employee numbers are 4-byte unsigned binary integers.

You pass an array as a single parameter. You must pass a table as multiple parameters, one parameter per column. Define an array in the template file as a single-column table. Define a table in the template file by specifying (after declaring the TABLE data type) the data type specification for each table column with a prefix of "C.". For example, the table of employee names and numbers described above would be defined in the template file as:

```
TABLE     6   INPUT
 C.CHAR  20   INPUT
 C.UBIN   4   INPUT
```

Note that the *data_length* field of the TABLE template is used to define the number of rows in the table. You can also specify a LEN parameter following the TABLE entry to allow dynamic specification of the number of rows at call time. For instance,

```
TABLE     *   INPUT
LEN       4   INPUT
 C.CHAR  20   INPUT
 C.UBIN   4   INPUT
```

defines a table with two columns. The number of rows is passed as the first parameter and the next two are the columns.

If the number of rows is variable, the number of rows for a particular call can be specified using one or two LEN parameters, depending on whether all of the columns contain only input values or if some of them contain output values:

- If no columns of the table are to contain output values, you only need to specify a single input LEN parameter for passing the number of rows in the input table.
- If any column of the table may contain output values, you can define either one or two LEN parameters:
  - If you define just one LEN parameter, it can be declared either for INPUT to specify the number of rows passed or OUTPUT to allow the called routine to return the number of rows used.
  - If you define two parameters, one must be declared for INPUT to specify the number of rows passed and the other must be declared for OUTPUT to allow the called routine to return the number of rows used.

Valid data directions are *, INPUT, OUTPUT, and INOUT. Specifications other than * are used for documentation or verification purposes only. The * specification indicates that no checking of column direction is to be done. The only exception to this is that an OUTPUT table may contain LEN or PTR data types with INPUT direction.

Specifying Lengths of Column Entries

The length of the elements in the table columns can be:

- Fixed and the same for each element
- Variable from call to call but the same for each element
- Variable from call to call and different for each element.

If the length is fixed and the same for each element, the length is specified with a *data_length* descriptor of *n*, where *n* is the length.

For example, a call with a single-column table of 4-byte signed binary input values and a variable number of rows would have the following template file declarations:

```
TABLE       *    INPUT    Table declaration
  LEN       4    INPUT    # of rows
  C.SBIN    4    INPUT    4-byte signed binary values
```

This defines two parameters in the parameter list: the first contains the number of rows and the second is an array of 4-byte signed binary values with the number of entries defined by the first parameter.

If variable from call to call but the same for all elements, a length parameter is required:

```
        C.BIT     *    INPUT    Bit string
         LEN      4    INPUT    Element length
```

This declaration defines a column of bit strings in which all the elements have the same length, which is specified in the following length parameter.

If variable and different for each element, the length descriptor can be either * or *n*, where *n* is the maximum length.

Note that, because all elements of a column *must* have the same maximum size, this case is most useful if the variable-length elements are indirectly addressed.

A C.LEN parameter is then declared as an array that contains the individual element lengths. This parameter is either INPUT or OUTPUT, depending on column definition. For example, let's look at a call with a three-column table with a variable number of rows. The first column will be an array of pointers to buffers where output values are to be placed. The second column is an array of binary values specifying the lengths of each individual buffer. The last column will contain the lengths of the values stored in the individual buffers by the called routine. The parameters would be described in the template file in the following manner:

```
      TABLE      *   INOUT    Table declaration
        LEN      4   INPUT    Max rows
        LEN      4   OUTPUT   Actual rows
        C.PTR    4   INPUT    Pointers to buffers
          CHAR   *   OUTPUT   Buffers are character
        C.LEN    4   INPUT    Maximum buffer lengths
        C.LEN    4   OUTPUT   Actual lengths used
```

This declaration defines five parameters in the parameter list:

- The number of buffers supplied (maximum rows)
- A variable to be set to the number of buffers used by the called routine
- An array of pointers to buffers
- An array of binary values specifying the individual buffer lengths
- An array of variables to contain the lengths of the values stored in the individual buffers

The associated parameter in REXX is a stemmed variable. If an output LEN parameter is declared for the associated TABLE definition, its value is stored in the *xxx*.0 stemmed variable (where *xxx* is the stemmed variable name) for any OUTPUT or INOUT type column parameters.

Any conversion required for specific data types is done by the REXX CSL interface (see the individual template data types for details).

Valid data directions are determined by the data type of the data contained in the array.

Pointer (Indirection)

Indirect data, where a pointer to the data is passed rather than the data itself, is declared using two entries in the template file. The first entry in the following example specifies a data type of PTR and a length of 4 bytes (the parameter type is 31-bit address). The second entry defines the type of the data pointed to and does not declare an actual parameter in the parameter list.

```
     PTR     4    INPUT
     CHAR   80    INPUT
```

Valid data types for indirection are:

- CHAR
- SBIN

- UBIN
- BIT
- LEN

Note that the FCHR, PTR, TABLE, RTNV, RTNC, and RTNR data types are not supported for indirection.

If the direction of the second template entry is INPUT, the direction of the first must also be INPUT. If the direction of the second is OUTPUT, the first also can be OUTPUT, indicating that the called routine will set it, or it can be INPUT, indicating that the address of the area for the output value is passed by the caller. The two entries together specify a single parameter, a 4-byte pointer to a data area of the format identified by the second declaration.

A LEN data type following a PTR data type does not specify the length of the pointer, but rather the length of the preceding parameter. For instance, suppose we had a table with 3 columns, the first an array of pointers to buffers to be filled by the calling routine, the second an array of pointers to 2-byte binary maximum length values, and the third an array of 4-byte signed binary elements to contain the actual length of each value placed in the buffer by the called routine. The template file definition would be:

```
      TABLE        *  INOUT    Table definition
        LEN        4  INPUT      Maximum number of rows
        LEN        4  OUTPUT     Used number of rows
        C.PTR      4  INPUT      Pointers to buffers
            CHAR   *  OUTPUT       Buffers
        C.PTR      4  INPUT      Pointers to buffer lengths
            LEN    2  INPUT        Maximum lengths
        C.LEN      4  OUTPUT     Used lengths for buffers
```

The call parameters for this template would be:

maxrows,usedrows,bufaddrs,lenaddrs,lengths

where:

maxrows
: indicates the maximum number of buffers usable

usedrows
: returns the number used on call completion

bufaddrs
: is an array of pointers to buffers with *maxrows* elements

lenaddrs
: is an array of pointers to 2-byte binary values specifying the lengths of the buffers

lengths
: is an array with *maxrows* elements, the first *usedrows* elements of which will be filled in by the called routine with the 4-byte binary lengths of the records placed in the corresponding buffers.

REXX does not support pointers. For REXX CSL calls, the actual variable name must be specified in the parameter list. The REXX CSL interface adds the pointer to the data for input parameters and uses the pointer to find the value to return in the variable for output parameters.

Defining Return Parameters for OPENVM Routines

This section describes how to define the return parameters used by routines in which OPENVM is specified on the CSLENTRY macro. Note that these data types are intended to be used when writing front-end routines for OPENVM CSL routines. The parameter list for OPENVM routines does not conform to the standard CSL parameter list structure. Use of these data types not only defines the length and direction of the return parameters, but also marks the location of the return parameters in the parameter list for later use by CSL.

Return Value Parameter

The return value parameter for an OPENVM routine is a signed binary integer. To declare the return value parameter, specify RTNV as the data type in the template file:

```
        RTNV     4     OUTPUT
```

The return value parameter always has a data length of 4 bytes and a data direction of OUTPUT. You must specify the length and direction in the template.

In REXX, the data type is numeric character string. The REXX CSL interface converts the binary value on output back to a numeric character string.

Return Code Parameter

The return code parameter for an OPENVM routine is a signed binary integer. To declare the return code parameter, specify RTNC as the data type in the template file:

```
        RTNC     4     OUTPUT
```

The return code parameter always has a data length of 4 bytes and a data direction of OUTPUT. You must specify the length and direction in the template.

In REXX, the data type is numeric character string. The REXX CSL interface converts the binary value on output back to a numeric character string.

Return Reason (Reason Code) Parameter

The return reason parameter for an OPENVM routine is a signed binary integer. To declare the return reason parameter, specify RTNR as the data type in the template file:

```
        RTNR     4     OUTPUT
```

The return reason parameter always has a data length of 4 bytes and a data direction of OUTPUT. You must specify the length and direction in the template.

In REXX, the data type is numeric character string. The REXX CSL interface converts the binary value on output back to a numeric character string.

Creating a Callable Services Library

After your routines are coded, you can create a CSL that is formatted for DASD or for a logical saved segment. The two steps involved are:

1. Building control files to show what routines go in the library
2. Issuing a command, CSLGEN, that uses the control files, text files, and template files to build the library and format it for storage.

Before building the library, note the following naming restrictions:

- Each routine within a callable services library must have a unique name. This is true whether the library will reside in a logical saved segment or on disk space.
- Direct call routines must not have the same name as the file name specified for the routine's text file.
- Direct call routines must not have the same name as the file name specified on any text record within the control file.
- Library names are restricted to a maximum of 8 characters.

  Note: Some high-level languages restrict the length allowed for external routine names. For example, FORTRAN only allows 7 characters for external routine names. The length of a routine name for a direct call routine should be tailored for the programming languages that may call it.

Creating CSL Control Files

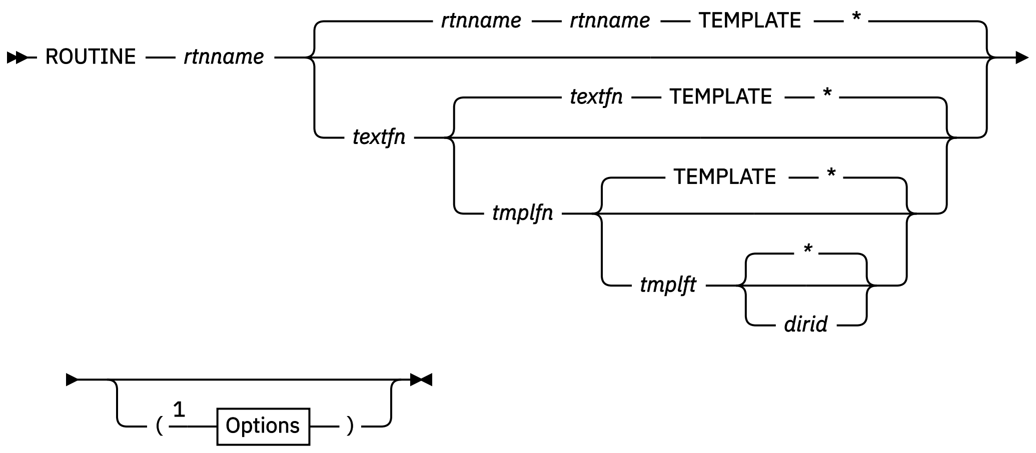
You can create a CSL control file using XEDIT. You can name this control file anything you want, although you may want to use a convention, such as making the file type CSLCNTRL.

A CSL control file consists of:

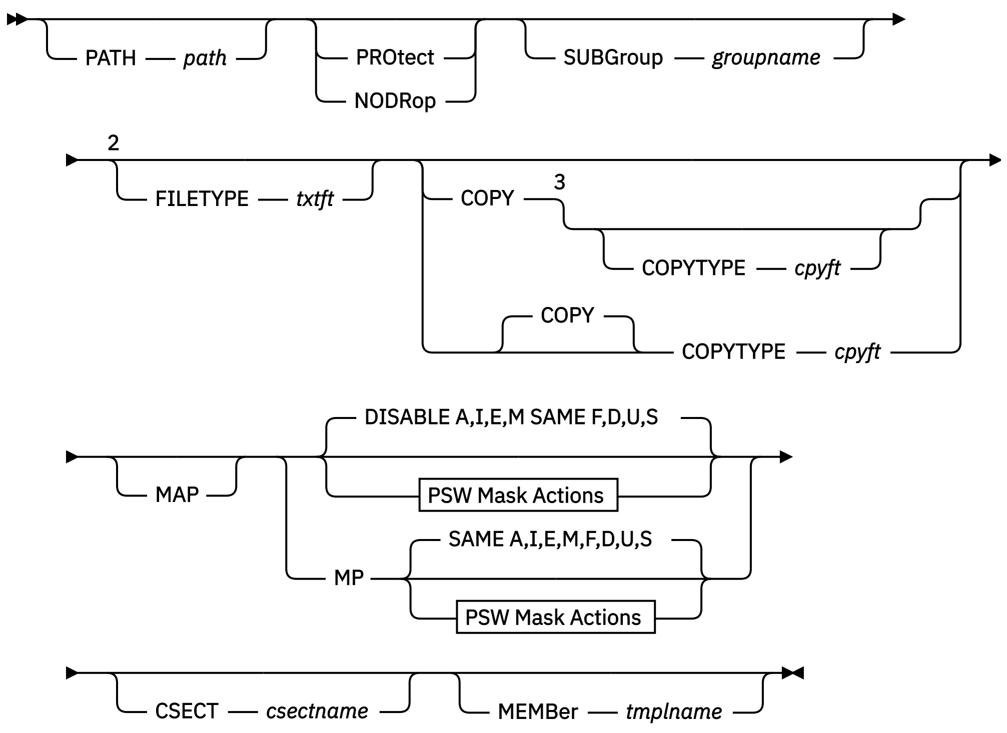
1. ROUTINE lines, which specify:
   - CSL routine names
   - ID of the program code associated with the routine name.
   - Template file ID
   - Whether the routine will use the direct or indirect CSL interface
   - The path used for the direct CSL interface
   - Whether the routine, when included in a segment-resident library, should be protected from removal after RTNLOADing
   - A library subgroup name which allows an application to RTNLOAD only a portion of a libraries routines at a time.
2. INCLUDE lines, which allow you to list text files which are linked along with the routine text file to complete the CSL routine object code.
3. ALIAS lines, which allow you to specify direct call alias names for use at RTNLOAD time. Defining an alias name using an ALIAS record allows you to define a path for the alias name, making it possible to RTNLOAD a direct call alias with a path different from the routine that has the alias name.
4. CSLCNTRL lines, which specify additional control file IDs, one per line.
5. TXTLIB lines, which list text libraries to be made global during the library build. TXTLIB lines can be used anywhere in the CSLCNTRL file to reorder the library search order.
6. TEXT lines, which list additional text files to be included in the CSL TXTLIB file.
7. Comment lines, denoted by an '*' as the first nonblank character in the line.

Note: The routine and alias names, assigned to routines specifying a path, must be unique within the CSLCNTRL file. They cannot match the file name specified on any text record or the file name specified for the text file on the same routine record.

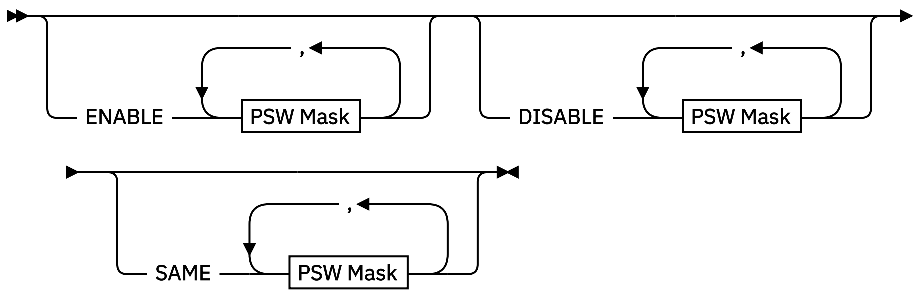
ROUTINE Line Format



Options



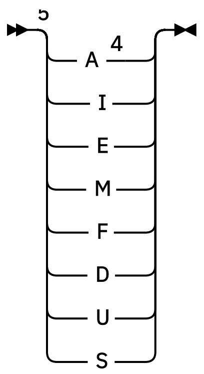
PSW Mask Actions



Notes:

[1] You can enter options in any order between the parentheses.
[2] The default is set by CSLGEN based upon the CSLGEN FILETYPE option.
[3] The default is set by CSLGEN based upon the CSLGEN COPYTYPE option.

PSW Mask



Notes:

where:

rtnname

is the name assigned to the routine that is to be included in the callable services library by CSLGEN. Each routine within a library must have a unique name.

If this routine is not directly callable then *rtnname* does not have to be the same as the TEXT file name (or TXTLIB member name). For instance, if all your TEXT file names must follow a certain naming convention, you could name the *rtnnames* differently so that they are easy to identify.

If this routine is to be directly callable (if it has a path specified) the name must be unique within the CSLCNTRL file and all CSLCNTRL files appended to it. The routine name cannot match:

- Any other routine name
- Any alias record routine name
- Any file name on a text record
- The *textfn* described below

textfn

is the name of a TEXT file (or TXTLIB member name) to be included in the callable services library as the routine identified by *rtnname*. If omitted, the default is the same as *rtnname*.

TEXT file names (and TXTLIB member names) are found and resolved by the CMS loader.

tmplfn

is the CMS file name of a CSL template file. If omitted, the default file name of the template file will be the same as the text file name specified in *textfn*.

tmplft

is the CMS file type of a CSL template file. If omitted, the default file type of the template file will be TEMPLATE.

dirid

identifies a directory or file mode where the template file is loaded. If omitted, the *dirid* of the template file defaults to ***, and the first file in the CMS search order that matches *tmplfn tmplft* will be taken.

PATH *path*

Specifies that the routine uses the direct CSL interface. The routine can also be invoked using the indirect CSL interface (CALL DMSCSL). The value of *path* can be expressed as;

p1.p2

specifying the two path tokens making up the path

*****

indicating that the path tokens *p1* and *p2.* are to be assigned by CSLGEN.

Possible values for *p1* are:

- An integer between 1 and 512 when the path can be shared by more than one routine. IBM reserves values 481 - 512 for its own use.
- An integer between 513 and 1024 when the path will be unique to *rtnname* while *rtnname* is loaded. IBM reserves values 993 - 1024 for its own use.

Any integer between 1 and 250 can be specified for *p2*.

When * is specified for *path* then *p1* is set to 1 and *p2* is generated based upon *rtnname*.

PROtect

specifies that after the initial RTNLOAD, subsequent attempts to RTNLOAD or RTNDROP this routine will not be allowed. Protected routines are removed when the segment they reside in is purged with a SEGMENT PURGE command. PROTECT is ignored by CSLGEN when creating a DASD CSL library.

NODrop

specifies that the routine cannot be dropped after it is loaded. However, the active version of the routine can be replaced by using the RTNLOAD command.

SUBGroup *groupname*
: labels the routine as part of a library subgroup. Subgroups can be loaded (using RTNLOAD) and dropped (using RTNDROP) at one time. This allows you to subset your library into several functional groups and RTNLOAD only those groups when they are needed.

FILETYPE *txtft*
: specifies a new file type, *txtft*, to be used when linking *textfn* together with the text files making up the CSL routine. When NOAUTO is specified on the CSLGEN command this option overrides the default file type CSLGEN sets using its own FILETYPE option. It overrides the default filetype only for the ROUTINE line upon which it appears. IF NOAUTO is not specified on the CSLGEN command then the FILETYPE option is ignored.

COPY
: saves a copy of the call routing code segment for this routine on the disk or directory specified by the 'TO dirid' option of the CSLGEN command. The code segment copy will have, by default, the file name *rtnname* and file type TEXT. The file type can be changed using the COPYTYPE option. This option is ignored if the PATH *path* is not specified.

COPYTYPE *cpyft*
: specifies that the file type of the call routing code segment, produced for the routine when the COPY option is specified, will be *cpyft*. This option overrides the default file type CSLGEN sets using its own COPYTYPE option. It overrides the default filetype only for the ROUTINE line upon which it appears. Specifying COPYTYPE *cpyft* implies the specification of COPY by default.

MAP
: Specifies that a load map file is to be created for the routine. This option is ignored if the SEG option is specified on the CSLGEN command.

  When specified the FULLMAP option is specified on the LOAD command used to link together the CSL routine object code.

  The load map file will have the name '*rtnname* LOADMAP'.

  The map file is saved on the same disk or directory that is specified for the library itself.

MP
: specifies that the routine is multiprocessor capable and does not require special call time preprocessing. This option can only be specified for a directly callable routine. A path, however, is not required. The MP option should not be specified if your CSL routine uses the CSENTRY macro. CSLENTRY does not provide multiprocessor support.

PSW_mask_actions
: is a list of actions to be performed on the Program Status Word (PSW) by the CSL interface before entering the CSL routine. After each action are listed the PSW masks to be altered by the action. Any mask not specified with an action receives the default action for that mask. The actions are:

  ENABLE
  : specifies that the listed masks must be enabled prior to entry of the CSL routine. Mask A (access register mode) cannot be specified with this action.

  DISABLE
  : specifies that the listed masks must be disabled prior to entry of the CSL routine. This is the default for masks A, I, E, and M if the MP keyword is *not* specified.

  SAME
  : specifies that the listed masks are not to be changed prior to entry of the CSL routine. This is the default for masks F, D, U, and S. This is also the default for masks A, I, E, and M if the MP keyword is specified.

  When the CSL routine is completed, the CSL interface returns the PSW to its original state before returning to the caller.

PSW_mask
: is a system interrupt or system mode that can be specified with a PSW mask action. A mask cannot be listed with more than one mask action. Any mask not specified with an action receives the default action for that mask. The masks are:

A
: Access register mode. The default action for this mask without the MP keyword is DISABLE. The default with the MP keyword is SAME. This mask cannot be specified with the ENABLE action.

I
: I/O interrupts. The default action for this mask without the MP keyword is DISABLE. The default with the MP keyword is SAME.

E
: External interrupts. The default action for this mask without the MP keyword is DISABLE. The default with the MP keyword is SAME.

M
: Machine check interrupts. The default action for this mask without the MP keyword is DISABLE. The default with the MP keyword is SAME.

F
: Fixed-point overflow program exception. The default action for this mask with or without the MP keyword is SAME.

D
: Decimal overflow program exception. The default action for this mask with or without the MP keyword is SAME.

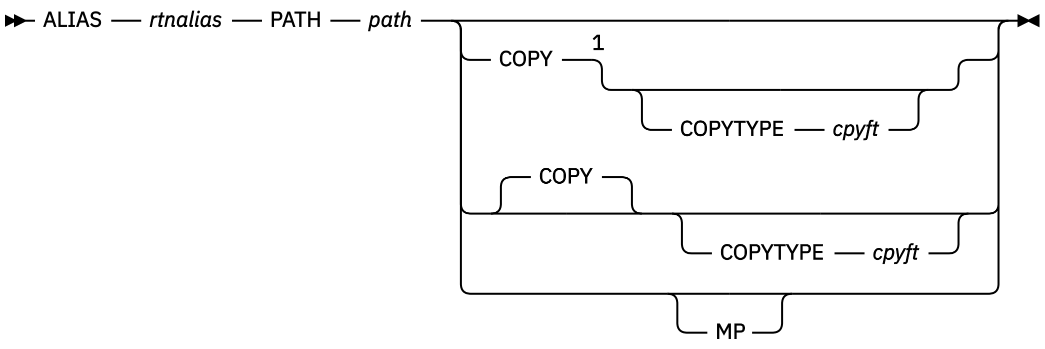
U
: Exponent underflow program exception. The default action for this mask with or without the MP keyword is SAME.

S
: Significance program exception. The default action for this mask with or without the MP keyword is SAME.

CSECT *csectname*
: specifies that the CSECT label in the text file *textfn* is not *textfn*. The actual CSECT label is specified as *csectname*. This option is intended for use where compatibility with existing routine and CSECT names makes creating a CSL routine which adheres to the existing restriction, that text file name and CSECT name be identical, impractical. This option should not be used unless it is absolutely necessary.

MEMBer *tmplname*
: specifies that the template file contains a collection of templates. The member of this template library to be used as the template for this routine is *tmplname*. This option allows many templates to be stored in one file.

ALIAS Line Format:



Notes:

[1] The default is set by CSLGEN based upon the CSLGEN COPYTYPE option.

where:

rtnalias
: is the alias name to be assigned. Many aliases may be defined for a single *rtnname* but each *rtnalias* must be unique within the library.

The *rtnalias* cannot be the same as any other routine name, alias name, template file name, or TEXT file name defined within the CSLCNTRL file.

The *rtnalias* must be distinct from any TEXT file name (or TXTLIB member name) in the library. *Rtnalias* cannot be the same as any other *rtnalias*, direct call routine name, or text record file name defined within the CSLCNTRL file or any appended CSLCNTRL file.

PATH *path*
specifies the path assigned for this alias name at RTNLOAD time. The direct routine RTNLOADed with this alias will be invoked using this path and the routine's own path definition ignored. The value of *path* can be expressed as:

p1.p2
specifying the two tokens making up the path or,

*****
indicating that the path tokens *p1* and *p2* are to be assigned by CSLGEN.

Possible values for *p1* are:

- An integer between 1 and 512 when the path can be shared by more than one routine. IBM reserves values 481 - 512 for its own use.
- An integer between 513 and 1024 when the path will be unique to *rtnname* while *rtnname* is loaded. IBM reserves values 993 - 1024 for its own use.

Any integer between 1 and 250 can be specified for *p2*.

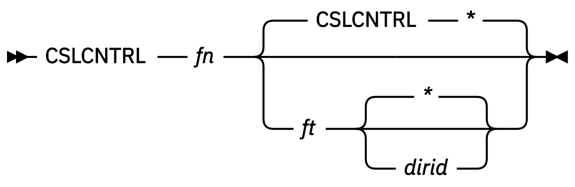
When * is specified for *path* then *p1* is set to 1 and *p2* is generated using *rtnalias*.

COPY
save a copy of the code routing segment for this alias on the disk or directory specified by the 'TO dirid' option of the CSLGEN command. The code segment copy will have, by default, the file name *rtnalias* and file type TEXT. The file type can be changed using the COPYTYPE option.

MP
specifies that *rtnalias* will be used for a multiprocessor capable routine. When MP is specified, *rtnalias* can only be used to alias routines specifying the MP option. When MP is not specified, *rtnalias* can only be used to alias routines not specifying the MP option.

COPYTYPE *cpyft*
specifies that the file type of the code routing code segment, produced for the routine when the COPY option is specified, will be *cpyft*. This option overrides the default filetype set by CSLGEN, based upon the CSLGEN COPYTYPE option. It overrides the default filetype only for the ALIAS line upon which it appears. Specifying COPYTYPE *cpyft* implies the specification of COPY by default.

CSLCNTRL Line Format



where:

fn
is the file name of another CSL control file.

ft
is the file type of another CSL control file.

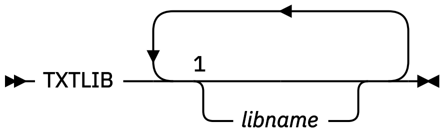
CSLCNTRL
specifies that the file type of another CSL control file is the default, CSLCNTRL.

dirid
identifies a directory or file mode where this additional control file resides.

*****
: specifies that the first file in the CMS file search order that matches the additional control file will be taken.

If specified, the additional control files may be used to identify additional routines to be included in the callable services library, and may specify additional CSL control files.

TXTLIB Line Format



Notes:

[1] A maximum of 63 repetitions.

where:

libname
: is the file name of up to 63 text libraries. The libraries are searched in the order they are named. If no *libname*s are specified, the GLOBAL TXTLIB environment is reset. For example, if you enter:

```
QUERY TXTLIB
```

The result is:

```
NONE
```

The default GLOBAL TXTLIB environment at CSL build time is

```
libname existing_global_environment CMSSAA
```

where:

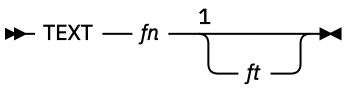
- *Libname* is the name of the library TXTLIB (if CSLGEN determines that this library needs to be created).
- *Existing_global_environment* is the list of TXTLIB names listed as the GLOBAL TXTLIB environment when the CSLGEN command was issued.

To override this default use a control file containing a TXTLIB record as its first record.

Note: The TXTLIB line may span several lines to hold all the necessary file names. A comma at the end of a TXTLIB record will indicate that the next record is a continuation of the current TXTLIB record. Each continuation TXTLIB record must begin with the word TXTLIB, the same as the first TXTLIB record.

TXTLIB and ROUTINE lines are processed in the same order as they are supplied in the CSLCNTRL file. A TXTLIB line anywhere in the control file sets the search order for loading the ROUTINE lines which follow.

TEXT Line Format:



Notes:

[1] The default is set by CSLGEN based upon the CSLGEN FILETYPE option when NOAUTO is specified. The default is TEXT when NOAUTO is not specified.

where:

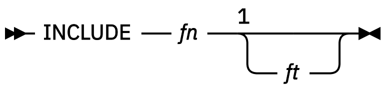
fn
: is the file name of a TEXT file to be included in the CSL library TXTLIB.

ft
: is the file type of the text file.

INCLUDE Line Format:

The INCLUDE line defines a text file that can reside on a disk or in a SFS directory and must be included to complete a CSL routine. The CSL routine it refers to is the routine defined by the last ROUTINE statement preceding the INCLUDE line. An INCLUDE line must be preceded by either a ROUTINE line or another INCLUDE line. As many INCLUDE lines may follow a ROUTINE line as are needed to list all of the text files to be included.

The line is ignored if NOAUTO is not specified on the CSLGEN command.



Notes:

[1] The default is set by CSLGEN based upon the CSLGEN FILETYPE option.

where:

fn
: is the file name of a TEXT file to be included in the CSL routine.

ft
: specifies that the file type of the text file to be included. It overrides the default filetype set by CSLGEN, based upon its own FILETYPE option. It overrides the default filetype only for the INCLUDE line it appears on.

Comment Line Format

Comment lines may be included anywhere in the control file. They must begin with an asterisk and be followed by at least one blank.

Building the Library

You can use the CSLGEN command to build a callable services library (CSL) from control files, text files, and template files. For more information on this command, see the *z/VM: CMS Commands and Utilities Reference*.

The LIBMAP and SEGMAP Files

Examples of the LIBMAP and SEGMAP files are shown using the MYLIB library example from the CSL assembler routine #2 example, found in the summary at the end of this section. The example's CSL library, MYLIB, is built using the following CSLCNTRL file.

```
ROUTINE TBSUM   TBSUMA  TBSUM   TEMPLATE A (PATH 513.1  Subgroup TBSUM
*
ALIAS   TBSORT  Path 513.4
*
ROUTINE TBSORT1 TBSORT1 TBSORT  TEMPLATE A (Subgroup TBSUM
ROUTINE TBSORT2 TBSORT2 TBSORT  TEMPLATE A (Subgroup TBSUM
ROUTINE TBSORT3 TBSORT3 TBSORT  TEMPLATE A (Subgroup TBSUM
ROUTINE TBSORT4 TBSORT4 TBSORT  TEMPLATE A (Subgroup TBSUM
ROUTINE TBSORT5 TBSORT5 TBSORT0 TEMPLATE A (Subgroup TBSUM
*
ROUTINE RTN1    CALC    CALC    TEMPLATE A (Subgroup TBSUM
*
```

Figure 26. CSLCNTRL File Example

Entering the command

```
    CSLGEN DASD MYLIB FROM MYLIB CSLCNTRL (MAP
```

would create a LIBMAP file similar to the following. The dates and times, of course, would differ.

```
TBSUM .000001 SD ******** 00000000 RMODE ANY AMODE 31
         TBSUMA TEXT A1                    6/14/92 11:41:08
TBSUMA SD 00010000 00000208 RMODE ANY AMODE 31
         TBSUMA TEXT A1                    6/14/92 11:41:08
      .000002 SD 00010208 00000000 RMODE ANY AMODE 31
         TBSORT TEXT A1                    6/15/92 14:31:50
TBSORT SD 00010208 00000040 RMODE ANY AMODE 31
         $$CSL$$ TXTLIB A1                 6/15/92 14:31:50
---------
TBSORT1 TBSORT1 SD 00010000 00000108 RMODE ANY AMODE 31
         TBSORT1 TEXT A1                 6/14/92 12:39:37
---------
TBSORT2 TBSORT2 SD 00010000 000005A8 RMODE ANY AMODE 31
         TBSORT2 TEXT A1                 6/14/92 12:57:29
---------
TBSORT3 TBSORT3 SD 00010000 00000324 RMODE ANY AMODE 31
         TBSORT3 TEXT A1                 6/14/92 12:45:30
---------
TBSORT4 TBSORT4 SD 00010000 000010AC RMODE ANY AMODE 31
         TBSORT4 TEXT A1                 6/16/92 13:01:25
---------
TBSORT5 TBSORT5 SD 00010000 000010AC RMODE ANY AMODE 31
         TBSORT5 TEXT A1                 6/15/92 13:59:25
---------
RTN1 CALC SD 00010000 00000132 RMODE ANY AMODE 31
         CALC TEXT A1                    6/13/92 11:10:34
---------
```

Figure 27. LIBMAP MYLIB

Routine TBSUM includes the direct call stub, TBSORT, which was created as a result of the TBSORT alias defined in the CSLCNTRL file. This stub is held in the temporary file $$CSL$$ TXTLIB. This file contains all of the newly created stub files and upon successful completion of the library build replaces the existing library text file. All GLOBAL TXTLIB commands issued by CSLGEN during processing use the $$CSL$$ in place of the library TXTLIB file to make sure that the lastest stubs are used in the library build.

The CSECT labels '.000001' and '.000002' were created by the assembler because of the placement of the REGEQU macro. The assembler, detecting the macro, created the CSECT label just in case executable code was generated by REGEQU.

Entering the command

```
    CSLGEN SEG MYLIB FROM MYLIB CSLCNTRL (MAP
```

would create a SEGMAP file similar to the following:

```
$$TMP$$ SD 00013000 00000348 RMODE ANY AMODE 31
       $$TMP$$ TEXT A1                      11/30/92 09:19:18
.000003 SD 00013348 00000000 RMODE ANY AMODE 31
       TBSUMA  TEXT A1                      6/14/92 11:41:08
TBSUM TBSUMA SD 00013348 00000208 RMODE ANY AMODE 31
       TBSUMA TEXT A1                       6/14/92 11:41:08
.000004 SD 00013550 00000000 RMODE ANY AMODE 31
       TBSORT  TEXT A1                      6/15/92 14:31:50
TBSORT SD 00013550 00000040 RMODE ANY AMODE 31
       $$CSL$$ TXTLIB A1                    6/15/92 14:31:50
TBSORT1 TBSORT1 SD 00013590 00000108 RMODE ANY AMODE 31
       TBSORT1 TEXT A1                      6/14/92 12:39:37
TBSORT2 TBSORT2 SD 00013698 000005A8 RMODE ANY AMODE 31
       TBSORT2 TEXT A1                      6/14/92 12:57:29
TBSORT3 TBSORT3 SD 00013C40 00000324 RMODE ANY AMODE 31
       TBSORT3 TEXT A1                      6/14/92 12:45:30
TBSORT4 TBSORT4 SD 00013F64 000010AC RMODE ANY AMODE 31
       TBSORT4 TEXT A1                      6/16/92 13:01:25
TBSORT5 TBSORT5 SD 00015010 000010AC RMODE ANY AMODE 31
       TBSORT5 TEXT A1                      6/15/92 13:59:25
RTN1 CALC SD 000160BC 00000132 RMODE ANY AMODE 31
       CALC TEXT A1                         6/13/92 11:10:34
$$CSL$$ SD 00013000 00000348 RMODE ANY AMODE 31
       $$CSL$$ TEXT A1                      11/30/92 09:19:18
```

Figure 28. SEGMAP MYLIB

Some libraries have two or more CSL routines which use the same text file and CSECT label. For example we can expand MYLIB as shown:

```
ROUTINE TBSUM   TBSUMA  TBSUM   TEMPLATE A (PATH 513.1  Subgroup TBSUM
*
ALIAS   TBSORT  Path 513.4
*
ROUTINE TBSORT1 TBSORT1 TBSORT  TEMPLATE A (Subgroup TBSUM
ROUTINE TBSORT2 TBSORT2 TBSORT  TEMPLATE A (Subgroup TBSUM
ROUTINE TBSORT3 TBSORT3 TBSORT  TEMPLATE A (Subgroup TBSUM
ROUTINE TBSORT4 TBSORT4 TBSORT  TEMPLATE A (Subgroup TBSUM
ROUTINE TBSORT5 TBSORT5 TBSORTO TEMPLATE A (Subgroup TBSUM
ROUTINE TBSORT6 TBSORT1 TBSORT  TEMPLATE A (Subgroup TBSUM
ROUTINE TBSORT7 TBSORT1 TBSORT  TEMPLATE A (Subgroup TBSUM
*
ROUTINE RTN1    CALC    CALC    TEMPLATE A (Subgroup TBSUM
*
```

Figure 29. MYLIB CSLCNTRL Expanded

After entering the CSLGEN command again, the LIBMAP file would look like the following example: Note that routines TBSORT1, TBSORT6, and TBSORT7 use the same text file, TBSORT1. To save file space, the maps of TBSORT6 and TBSORT7 are not appended to the load map. The names of CSL routines are listed in a vertical column preceding the load map. CSL routines which have the same load map are displayed in this manner.

```
TBSUM .000016 SD ******** 00000000 RMODE ANY AMODE 31
      TBSUMA TEXT A1                        6/14/92 11:41:08
TBSUMA SD 00010000 00000208 RMODE ANY AMODE 31
      TBSUMA TEXT A1                        6/14/92 11:41:08
.000017 SD 00010208 00000000 RMODE ANY AMODE 31
      TBSORT TEXT A1                        6/15/92 14:31:50
TBSORT SD 00010208 00000040 RMODE ANY AMODE 31
      $$CSL$$ TXTLIB A1                     6/15/92 14:31:50
---------
TBSORT7
TBSORT6
TBSORT1 TBSORT1 SD 00010000 00000108 RMODE ANY AMODE 31
      TBSORT1 TEXT A1                       6/14/92 12:39:37
---------
TBSORT2 TBSORT2 SD 00010000 000005A8 RMODE ANY AMODE 31
      TBSORT2 TEXT A1                       6/14/92 12:57:29
---------
TBSORT3 TBSORT3 SD 00010000 00000324 RMODE ANY AMODE 31
      TBSORT3 TEXT A1                       6/14/92 12:45:30
---------
TBSORT4 TBSORT4 SD 00010000 000010AC RMODE ANY AMODE 31
      TBSORT4 TEXT A1                       6/16/92 13:01:25
---------
TBSORT5 TBSORT5 SD 00010000 000010AC RMODE ANY AMODE 31
      TBSORT5 TEXT A1                       6/15/92 13:59:25
---------
RTN1 CALC SD 00010000 00000132 RMODE ANY AMODE 31
      CALC TEXT A1                          6/13/92 11:10:34
---------
```

Figure 30. LIBMAP MYLIB

In the SEGMAP file, multiple CSL routines using the same CSECT label are listed vertically before the CSECT label in the SEGMAP file.

```
$$TMP$$ SD 00013000 00000348 RMODE ANY AMODE 31
       $$TMP$$ TEXT A1                      11/30/92 09:19:18
.000018 SD 00013348 00000000 RMODE ANY AMODE 31
       TBSUMA TEXT A1                       6/14/92 11:41:08
TBSUM TBSUMA SD 00013348 00000208 RMODE ANY AMODE 31
       TBSUMA TEXT A1                       6/14/92 11:41:08
.000019 SD 00013550 00000000 RMODE ANY AMODE 31
       TBSORT TEXT A1                       6/15/92 14:31:50
TBSORT SD 00013550 00000040 RMODE ANY AMODE 31
       $$CSL$$ TXTLIB A1                    6/15/92 14:31:50
TBSORT7
TBSORT6
TBSORT1 TBSORT1 SD 00013590 00000108 RMODE ANY AMODE 31
       TBSORT1 TEXT A1                      6/14/92 12:39:37
TBSORT2 TBSORT2 SD 00013698 000005A8 RMODE ANY AMODE 31
       TBSORT2 TEXT A1                      6/14/92 12:57:29
TBSORT3 TBSORT3 SD 00013C40 00000324 RMODE ANY AMODE 31
       TBSORT3 TEXT A1                      6/14/92 12:45:30
TBSORT4 TBSORT4 SD 00013F64 000010AC RMODE ANY AMODE 31
       TBSORT4 TEXT A1                      6/16/92 13:01:25
TBSORT5 TBSORT5 SD 00015010 000010AC RMODE ANY AMODE 31
       TBSORT5 TEXT A1                      6/15/92 13:59:25
RTN1 CALC SD 000160BC 00000132 RMODE ANY AMODE 31
       CALC TEXT A1                         6/13/92 11:10:34
$$CSL$$ SD 00013000 00000348 RMODE ANY AMODE 31
       $$CSL$$ TEXT A1                      11/30/92 09:19:18
```

Figure 31. SEGMAP MYLIB

Building a Library Using Alternate TEXT File Types

The way to effectively build CSL libraries using alternate file types is to understand how the NOAUTO option of the CSLGEN command alters CSLGEN processing. CSL routines are packaged into a DASD CSL library as individual modules. The segment version of a CSL library is one large MODULE file. To build these modules, CSLGEN uses the CMS loader to link together the various text files and TXTLIB members needed to complete the CSL routine object code. The NOAUTO option causes CSLGEN to take greater control of the LOAD processing.

When NOAUTO is not specified, CSLGEN allows the CMS loader to resolve any unresolved external references it may have by first searching for a suitable text file (file type TEXT) in accessed DASD and SFS directories. If a suitable text file is not found, then the loader searches any TXTLIB files currently globaled. CSLGEN ignores any INCLUDE statements following a ROUTINE statement since it has no control over what is included by the loader.

When NOAUTO is specified, CSLGEN uses the INCLUDE statements following a ROUTINE statement to issue a sequence of LOAD and INCLUDE commands to tell the loader to use text files with a defined file type. Automatic searching to resolve external references is postponed until all of the text files listed with INCLUDE statements have been included using INCLUDE commands. Automatic searching of DASD and SFS directory text files is never performed to resolve external references.

The following example uses routines from the VMLIB library and requires that the CMS object disk be accessed. By default that is MAINT*vrm* 3B2. Assume that the object disk has been accessed as file mode H. We'll create a CSLCNTRL file called EXAMPLE CSLCNTRL to hold these two routines.

```
  TXTLIB  VMMTLIB
*
  ROUTINE DMSRENAM DMSJRN   DMSJRN   TEMPLATE *
*
  ROUTINE CMCFMDZ  DMSACD   DMSACD   (MP PATH 1024.023
```

Figure 32. CSLCNTRL Example

Entering the command

```
       CSLGEN DASD EXAMPLE FROM EXAMPLE (MAP
```

results in the following LIBMAP file.

```
DMSRENAM DMSJRN SD 01100000 000014E8 RMODE ANY AMODE 31
      DMSJRN TEXT H           1             2/03/93 09:27:57
 ----------
CMCFMDZ DMSACD SD 01100000 00000738 RMODE ANY AMODE 31
      DMSACD TEXT H           1             2/03/93 09:28:58
DMSACL SD 01100738 00000190 RMODE ANY AMODE 31
      DMSACL TEXT H           1             2/03/93 09:30:31
DMSAID SD 011008C8 000003B0 RMODE ANY AMODE 31
      DMSAID TEXT H           1             2/03/93 09:30:31
DMSAWD SD 01100C78 000001E0 RMODE ANY AMODE 31
      DMSAWD TEXT H           1             2/03/93 09:30:33
VMMUXAC SD 01100E58 00000040 RMODE ANY AMODE ANY
      VMMTLIB TXTLIB S        2 VMMUXAC     2/03/93 04:29:12
VMMUXRE SD 01100E98 00000040 RMODE ANY AMODE ANY
      VMMTLIB TXTLIB S        2 VMMUXRE     2/03/93 04:29:12
VMQUESE SD 01100ED8 00000040 RMODE ANY AMODE ANY
      VMMTLIB TXT LIB S       2 VMQUESE     2/03/93 04:29:12
----------
```

Figure 33. LIBMAP Omitting NOAUTO

Entering the command

```
CSLGEN DASD EXAMPLE FROM EXAMPLE (MAP NOAUTO REPLACE
```

results in the following error. No searching of DASD or SFS directories is performed except for those text files listed in the control file.

```
 DMSLIO201W The following names are undefined:
  DMSACL   DMSAID   DMSAWD
 DMSWCG1110E CSLGEN encountered an error executing LOAD, RC=4
 DMSWCG1109I CSLGEN terminated. No library built.
```

To use the NOAUTO option you must, as a minimum, list all of the text files which you expect to be found only on DASD or SFS directories. To demonstrate this, see the following modified EXAMPLE CSLCNTRL:

```
  TXTLIB  VMMTLIB
*
  ROUTINE DMSRENAM DMSJRN   DMSJRN   TEMPLATE *
*
  ROUTINE CMCFMDZ  DMSACD   DMSACD   (MP PATH 1024.023
    INCLUDE DMSAWD
    INCLUDE DMSAID
    INCLUDE DMSACL
```

Figure 34. CSLCNTRL with INCLUDE Statements

Again, entering the command

```
  CSLGEN DASD EXAMPLE FROM EXAMPLE (MAP NOAUTO REPLACE
```

results in successful completion and the LIBMAP file looks as follows:

```
DMSRENAM DMSJRN SD 01100000 000014E8 RMODE ANY AMODE 31
        DMSJRN TEXT H1                      2/08/93 12:36:25
----------
CMCFMDZ DMSACD SD 01100000 00000738 RMODE ANY AMODE 31
        DMSACD TEXT H1                      2/08/93 12:37:40
DMSAWD SD 01100738 000001E0 RMODE ANY AMODE 31
        DMSAWD TEXT H1                      2/08/93 12:57:20
DMSAID SD 01100918 000003B0 RMODE ANY AMODE 31
        DMSAID TEXT H1                      2/08/93 12:57:49
DMSACL SD 01100CC8 00000190 RMODE ANY AMODE 31
        DMSACL TEXT H1                      2/08/93 12:58:19
VMMUXAC SD 01100E58 00000040 RMODE ANY AMODE ANY
        VMMTLIB TXTLIB S2         VMMUXAC   2/06/93 07:55:53
VMMUXRE SD 01100E98 00000040 RMODE ANY AMODE ANY
        VMMTLIB TXTLIB S2         VMMUXRE   2/06/93 07:55:53
VMQUESE SD 01100ED8 00000040 RMODE ANY AMODE ANY
        VMMTLIB TXTLIB S2         VMQUESE   2/06/93 07:55:53
----------
```

Figure 35. LIBMAP with NOAUTO and INCLUDE

Using the INCLUDE statements you not only were able to include the three text files, DMSAWD, DMSAID, and DMSACL but you were also able to alter the order in which they were included. The sequence of INCLUDE commands are issued by CSLGEN in the same order as the INCLUDE list in the control file. Using the modified CSLCNTRL file for EXAMPLE, let us see what the LOAD and INCLUDE command sequence issued by CSLGEN is.

If you enter

```
       CSLGEN DASD EXAMPLE FROM EXAMPLE (MAP
```

the sequence of load commands are:

```
LOAD DMSJRN (RLDSAVE NCHIST  FULLMAP
                  .
                  .       (Additional processing is performed
                  .        between these LOAD commands to
                  .        create a module of DMSRENAM)
                  .

LOAD DMSACD (RLDSAVE NCHIST  FULLMAP
```

Figure 36. LOAD Sequence When NOAUTO is Not Used

If you enter the command

```
       CSLGEN DASD EXAMPLE FROM EXAMPLE (MAP NOAUTO
```

the sequence of LOAD and INCLUDE commands issued by CSLGEN are:

```
LOAD DMSJRN (RLDSAVE NCHIST NOAUTO LIBE UNDEF
          FULLMAP FILETYPE TEXT
              .
              .       (Additional processing is preformed
              .        between these LOAD commands to
              .        create a module of DMSRENAM)
              .
.
LOAD DMSACD (RLDSAVE NCHIST NOAUTO NOLIBE NOUNDEF
          FULLMAP FILETYPE TEXT
INCLUDE  DMSAWD ( NOAUTO NOLIBE NOUNDEF FULLMAP FILETYPE TEXT
INCLUDE  DMSAID ( NOAUTO NOLIBE NOUNDEF FULLMAP FILETYPE TEXT
INCLUDE  DMSACL ( NOAUTO LIBE UNDEF FULLMAP FILETYPE TEXT
```

Figure 37. LOAD Sequence When NOAUTO is Used

You can see from the command sequence of the preceding figure, the FILETYPE option is specified on each LOAD or INCLUDE command. The FILETYPE options on both the CSLGEN command and the

ROUTINE statement as well as the file type operand on the INCLUDE statement allow you to change what file type is specified on the CMS LOAD and INCLUDE commands entered.

If you copy all of the text files used in EXAMPLE, giving them file types of TXT, and entering the command

```
CSLGEN DASD EXAMPLE FROM EXAMPLE (MAP FILETYPE TXT REPLACE
```

results in the following LOAD and INCLUDE sequences. Note that the use of the FILETYPE option on CSLGEN also implies NOAUTO.

```
LOAD DMSJRN (RLDSAVE NCHIST NOAUTO LIBE UNDEF FULLMAP
         FILETYPE TXT
              .
              .       (Additional processing is preformed
              .        between these LOAD commands to
              .        create a module of DMSRENAM)
                     .

LOAD DMSACD (RLDSAVE NCHIST NOAUTO NOLIBE NOUNDEF
        FULLMAP FILETYPE TXT
INCLUDE  DMSAWD ( NOAUTO NOLIBE NOUNDEF FULLMAP FILETYPE TXT
INCLUDE  DMSAID ( NOAUTO NOLIBE NOUNDEF FULLMAP FILETYPE TXT
INCLUDE  DMSACL ( NOAUTO LIBE UNDEF FULLMAP FILETYPE TXT
```

Figure 38. LOAD Sequence When FILETYPE TXT is Used

The following LIBMAP is also produced:

```
DMSRENAM DMSJRN SD 01100000 000014E8 RMODE ANY AMODE 31
          DMSJRN TXT A1                    2/08/93 12:36:25
----------
CMCFMDZ DMSACD SD 01100000 00000738 RMODE ANY AMODE 31
          DMSACD TXT A1                    2/08/93 12:37:40
DMSAWD SD 01100738 000001E0 RMODE ANY AMODE 31
          DMSAWD TXT A1                    2/08/93 12:57:20
DMSAID SD 01100918 000003B0 RMODE ANY AMODE 31
          DMSAID TXT A1                    2/08/93 12:57:49
DMSACL SD 01100CC8 00000190 RMODE ANY AMODE 31
          DMSACL TXT 1                     2/08/93 12:58:19
VMMUXAC SD 01100E58 00000040 RMODE ANY AMODE ANY
          VMMTLIB TXTLIB S2       VMMUXAC   2/06/93 07:55:53
VMMUXRE SD 01100E98 00000040 RMODE ANY AMODE ANY
          VMMTLIB TXTLIB S2       VMMUXRE   2/06/93 07:55:53
VMQUESE SD 01100ED8 00000040 RMODE ANY AMODE ANY
          VMMTLIB TXTLIB S2       VMQUESE   2/06/93 07:55:53
----------
```

Figure 39. LIBMAP with FILETYPE TXT

If you modify EXAMPLE CSLCNTRL with the use of file types you alter the file type used for those selected text files.

```
  TXTLIB  VMMTLIB
*
  ROUTINE DMSRENAM DMSJRN DMSJRN TEMPLATE *
*
  ROUTINE CMCFMDZ DMSACD DMSACD (MP PATH 1024.023
            FILETYPE TEXT
    INCLUDE DMSAWD
    INCLUDE DMSAID TEXT
    INCLUDE DMSACL
```

Figure 40. CSLCNTRL Example with File Types Specified

You once again enter the command

```
  CSLGEN DASD EXAMPLE FROM EXAMPLE (MAP FILETYPE TXT REPLACE
```

using the modified EXAMPLE control file. The use of file types in the specified ROUTINE and INCLUDE statements results in the following LOAD and INCLUDE sequences. Note that the use of the file types in the CSLCNTRL file changes the LOAD and INCLUDE FILETYPE option for the text file specified on the line containing the file type. No other LOAD or INCLUDE statements are affected.

```
LOAD DMSJRN (RLDSAVE NCHIST NOAUTO LIBE UNDEF FULLMAP
       FILETYPE TXT
              .
              .       (Additional processing is preformed
              .        between these LOAD commands to
              .        create a module of DMSRENAM)
              .
              .
LOAD DMSACD (RLDSAVE NCHIST NOAUTO NOLIBE NOUNDEF FULLMAP
      FILETYPE TEXT
INCLUDE  DMSAWD ( NOAUTO NOLIBE NOUNDEF FULLMAP FILETYPE TXT
INCLUDE  DMSAID ( NOAUTO NOLIBE NOUNDEF FULLMAP FILETYPE TEXT
INCLUDE  DMSACL ( NOAUTO LIBE UNDEF FULLMAP FILETYPE TXT
```

Figure 41. LOAD Sequence File Types in the Control File Are Used

The following LIBMAP is also produced.

```
DMSRENAM DMSJRN SD 01100000 000014E8 RMODE ANY AMODE 31
         DMSJRN TXT A1                      2/08/93 12:36:25
----------
CMCFMDZ DMSACD SD 01100000 00000738 RMODE ANY AMODE 31
         DMSACD TEXT H1                     2/08/93 12:37:40
DMSAWD SD 01100738 000001E0 RMODE ANY AMODE 31
         DMSAWD TXT A1                      2/08/93 12:57:20
DMSAID SD 01100918 000003B0 RMODE ANY AMODE 31
         DMSAID TEXT H1                     2/08/93 12:57:49
DMSACL SD 01100CC8 00000190 RMODE ANY AMODE 31
         DMSACL TXT A1                      2/08/93 12:58:19
VMMUXAC SD 01100E58 00000040 RMODE ANY AMODE ANY
         VMMTLIB TXTLIB S2        VMMUXAC   2/06/93 07:55:53
VMMUXRE SD 01100E98 00000040 RMODE ANY AMODE ANY
         VMMTLIB TXTLIB S2        VMMUXRE   2/06/93 07:55:53
VMQUESE SD 01100ED8 00000040 RMODE ANY AMODE ANY
         VMMTLIB TXTLIB S2        VMQUESE   2/06/93 07:55:53
----------
```

Figure 42. LIBMAP with FILETYPE TXT

Using INCLUDE Statements to Substitute TXTLIB Members

In the previous section it was shown that to build a library using the NOAUTO option each ROUTINE statement must be followed, at a minimum, by INCLUDE statements listing all of the additional linking text files that reside on DASD or SFS directories. TXTLIB members do not require listing.

However if it is necessary to allow substitution, at build time, of TXTLIB members, an INCLUDE statement is necessary for each TXTLIB member to be substituted. This is different from the way CSLGEN handles substitutions when NOAUTO is not specified.

To demonstrate, use the following version of EXAMPLE CSLCNTRL. Also, assume that text files VMMUXAC TEXT, VMMUXRE TEXT, and VMQUESE TEXT are on file mode 'A' so that text files can be substituted in place of the TXTLIB members.

```
TXTLIB  VMMTLIB
*
ROUTINE DMSRENAM DMSJRN   DMSJRN   TEMPLATE *
*
ROUTINE CMCFMDZ  DMSACD   DMSACD   (MP PATH 1024.023
```

Figure 43. CSLCNTRL

Entering the command

```
CSLGEN DASD EXAMPLE FROM EXAMPLE (MAP REPLACE
```

results with the following LIBMAP file. The LIBMAP file shows the substitution of the text files for the TXTLIB members.

```
DMSRENAM DMSJRN SD 01100000 000014E8 RMODE ANY AMODE 31
         DMSJRN TEXT H1                    2/03/93 09:27:57
----------
CMCFMDZ DMSACD SD 01100000 00000738 RMODE ANY AMODE 31
         DMSACD TEXT H1                    2/03/93 09:28:58
DMSACL SD 01100738 00000190 RMODE ANY AMODE 31
         DMSACL TEXT H1                    2/03/93 09:30:31
DMSAID SD 011008C8 000003B0 RMODE ANY AMODE 31
         DMSAID TEXT H1                    2/03/93 09:30:31
DMSAWD SD 01100C78 000001E0 RMODE ANY AMODE 31
         DMSAWD TEXT H1                    2/03/93 09:30:33
VMMUXAC SD 01100E58 00000040 RMODE ANY AMODE ANY
         VMMUXAC TEXT A1                   2/03/93 04:29:12
VMMUXRE SD 01100E98 00000040 RMODE ANY AMODE ANY
         VMMUXRE TEXT A1                   2/03/93 04:29:12
VMQUESE SD 01100ED8 00000040 RMODE ANY AMODE ANY
         VMQUESE TEXT A1                   2/03/93 04:29:12
-------
```

Figure 44. LIBMAP with NOAUTO

The figure below shows the control file that would be required to obtain the same results using the NOAUTO option.

```
TXTLIB  VMMTLIB
*
  ROUTINE DMSRENAM DMSJRN   DMSJRN   TEMPLATE *
*
  ROUTINE CMCFMDZ  DMSACD   DMSACD   (MP PATH 1024.023
    INCLUDE DMSACL
    INCLUDE DMSAID
    INCLUDE DMSAWD
    INCLUDE VMMUXAC
    INCLUDE VMMUXRE
    INCLUDE VMQUESE
```

Figure 45. CSLCNTRL with INCLUDE Statements

Determining when to use INCLUDE statements is important since each additional INCLUDE statement requires additional CSLGEN processing as well as the execution of the INCLUDE command. For large libraries with many text files (for example, VMLIB) the use of INCLUDE statements to list all text files for all library routines may cause a serious degradation in CSLGEN performance.

Protecting Routines

The protected routine concept provides the ability to prevent front-ending critical CSL routines.

The protection of a routine is determined at CSLGEN time. When you perform a CSLGEN command to create a segment resident library, a routine is marked protected if you specify PROTECT on the ROUTINE records for the CSLCNTRL file. The protected routine, once loaded will remain unchangeable through RTNLOAD and RTNDROP until the segment is purged.

A routine can be protected from removal but still be replaced as the active version by specifying NODROP on the ROUTINE record. A drop-protected routine may be front-ended, whereas a protected routine cannot. This lock can be applied to routines that reside in DASD-resident or segment-resident libraries. This lock is most useful when protecting critical front-end routines for OPENVM calls, because some routines use no return value or return code parameters. If these routines are dropped, subsequent calls to them will result in ABENDs.

Path Description and Choosing a Path

A path is composed of a pair of positive integers. It directs the call routing code segment in it's traversal of internal CSL tables to find the proper CSL routine. The syntax of the path is denoted as (p1.p2).

The p1 integer is broken into two groups:

1. 1 - 512 — Paths starting with one of these numbers are termed shared paths. Each path references a list of routines concurrently sharing that path. These paths provide better performance than the DMSCSL interface because the list of routines examined is a smaller list than that used by DMSCSL.

   The primary advantage of these paths is that a unique path need not be maintained for each routine. This reduces the need for path assignment coordination to insure that CSL routines do not use the same unique path at the same time.

   IBM reserves values 481 - 512 for its own use.
2. 513 - 1024 — Paths starting with one of these numbers are referred to as unique paths. The path directly addresses a single routine entry point. The path is locked to the run name specified at RTNLOAD time. You cannot PUSH a new routine using a unique path on top of the currently active routine, using the same path, if the run name you specify differs from the run name of the currently active routine. The user must first RTNDROP all routine versions using the original path and run name before the new run name may be loaded. These paths provide the best CALL time performance.

   IBM reserves values 993 - 1024 for its own use. CMS reserves values 1022, 1023, and 1024 for use by routines in its VMLIB and VMMTLIB CSL libraries.

Close attention to unique path assignments is critical. An example of a situation which can produce undesirable effects when unique paths are mismanaged follows:

A CSL routine named 'A', which uses a unique path is loaded. Later a program calls a routine, 'B', which uses the same path as 'A' while the path is still active for 'A'. The call routing code segment for 'B' will traverse the same path defined for 'A' resulting in routine 'A' being executed.

There are several ways to arrive at this situation. The result, however, will be the same. The direct call of 'B' will in reality pass control to routine 'A'. Any routine name which uses the same path as 'A' in it's call routing code segment could be considered an alias of 'A' when using direct calls.

Library Subgroups

The library subgroup is an extension of the existing GROUP option used in RTNLOAD and RTNDROP. A large library can be subdivided into functionally related subsets using the library subgroup option. Such groupings can save storage and improve call time performance.

Using Template Libraries in Place of Individual Template Files

To avoid the need to keep many individual template files, you may create a template library that contains one or more template members. If the MEMBER option is specified on the ROUTINE line in the CSL control file, CSLGEN accesses the specified member of the template library when building the routine.

Within the template library, template members are separated and identified by an identification line. This line is a noncomment line having the keyword TEMPLATE as the first nonblank string on the line. Following the TEMPLATE keyword is a label that identifies the template.

For example, consider the file TEST TEMPLATE, which contains the templates shown in Figure 46 on page 263.

```
************************************************************
TEMPLATE ALPHA
*
* This is template alpha. The individual data type
* definitions have been offset to improve readability.
*
  OPENVM 8 7            7 parameters, all required
  SBIN    4  INPUT      File_descriptor
  PTR     4  INPUT      Buffer address pointer
   CHAR   *  INOUT      Contents of Buffer
  SBIN    4  INPUT      Buffer_ALET
  SBIN    4  INPUT      Read_count/Write_count
  RTNV    4  OUTPUT     Return_Value
  RTNC    4  OUTPUT     Return_Code
  RTNR    4  OUTPUT     Reason_Code
*
************************************************************
*
TEMPLATE BETA
*
*
  OPENVM 2 2       2 parameter template  2 req parameters
  UBIN   4 INPUT   Handle to String (that had left side
                   appended)
  UBIN   4 INPUT   Handle to String (that had right side
                   appened)
*
*
************************************************************
*
TEMPLATE beta
*
* Since the template search performed by CSLGEN is case
* sensitive the label 'beta' identifies a different
* member than 'BETA'.
*
  OPENVM 3 3       3 parameter template   3 req parameters
  RTNC   4 OUTPUT  Return code
  UBIN   4 INPUT   Handle to String
                   (that had left side appended)
  UBIN   4 INPUT   Handle to String (that had right
                   side appened)
*
***********************************************************
```

Figure 46. Example of Template Library (TEST TEMPLATE)

If we want to use the beta template to define the parameter list for a routine called BETA5, we could do so with the following ROUTINE line.

```
ROUTINE BETA5 BETA5 TEST TEMPLATE A (MEM beta
```

CSLGEN would retrieve the beta template from TEST TEMPLATE and use it to build BETA5.

Invoking CSL Routines

Application programs must invoke CSL routines using the proper assembler interface for their respective languages. Programs invoke CSL routines with a call to DMSCSL. If an application program written in assembler invokes a CSL routine multiple times, a *fast path* (CSLFPI macro) is available.

The following example shows an assembler program calling the CSL routine, DMSOPEN.

```
         .
     .
     .
     CALL  DMSCSL,(ROUTINE,RETURN,REASON,PARM1,PARM2,
           PARM3,PARM4,PARM5),VL
     .
     .
     .
ROUTINE  DC    C'DMSOPEN '
RETURN   DC    F'0'
REASON   DC    F'0'
PARM1    DC    C'TESTMC FILE .'
PARM2    DC    A(L'PARM1)
```

```
PARM3    DC    C'READ CACHE'
PARM4    DC    A(L'PARM3)
PARM5    DC    C'          '
```

For more information about invoking CSL routines, either your own or those supplied in VMLIB, see the *z/VM: CMS Application Development Guide*. Included in the description is how to invoke CSL routines frequently from Assembler programs.

CSL Summary and Example

These steps summarize what you must do to create routines and build callable services libraries from those routines:

1. Code the CSL routine in assembler language and then assemble it to make a TEXT deck.
2. Create a template file for the routine, using XEDIT, that describes the routine's parameters, or add the template to a template library.

   Note: You must repeat steps "1" on page 264 and "2" on page 264 for every routine that you want to reside in a callable services library (CSL).
3. Create a control file, using XEDIT, that describes the routines that are to be placed in the CSL.
4. Using the CSLGEN command and the control file created in step "3" on page 264, build a callable services library that will reside on DASD or in a saved segment.
5. Using the GLOBAL CSLLIB command and RTNLOAD commands, make the library routines accessible to calls.

The following example illustrates these steps: it shows two example CSL routines coded in assembler and template files for each, the control file, the commands necessary to build a library and then make it accessible, and finally invocations from example programs.

CALC: Example CSL Assembler Routine #1

The first example CSL routine is called CALC. It is not a directly callable routine.

CALC ASSEMBLE

```
************************************************************
*                                                          *
*  Routine name:                                           *
*         CALC                                             *
*                                                          *
*  Function:                                               *
*         Adds the first two numbers passed to it.   If    *
*         a third number is passed, divides the  result    *
*         by it.  If less than two numbers are  passed,    *
*         DMSCSL will return a return code of -11.   If    *
*         more  than  three  are  passed,  DMSCSL  will    *
*         return a return code of -10.                     *
*                                                          *
*  Parameter list:                                         *
*         Return code                 Fullword             *
*         Result field                Fullword             *
*         1st operand                 Fullword             *
*         2nd operand                 Fullword             *
*         3rd operand (optional)      Fullword             *
*                                                          *

************************************************************
         SPACE 1
************************************************************
*  Register Equates
************************************************************
         REGEQU
         EJECT
************************************************************
*  Module Entry Logic
************************************************************
         SPACE 1
CALC     CSLENTRY (RETURN,RESULT,OP1,OP2,DIVISOR)
```

```
         LR    R2,R1                  Save address of PLIST
         LR    R3,R0                  Save number of operands
         SPACE 1

************************************************************
*  Main Program Function Logic
************************************************************
         SPACE
FUNCTION DS    0H
         CSLGETP PLIST=(R2),     Get 1st operand address   X
               PARM=OP1,                                   X
               ADDRESS=(R5)
         CSLGETP PLIST=(R2),     Get 2nd operand address   X
               PARM=OP2,                                   X
               ADDRESS=(R6)
         SPACE 1

         L     R5,0(,R5)         Get 1st operand
         A     R5,0(,R6)         Add 2nd operand
         SPACE 1
         CH    R3,=H'4'          Optional operand present?
         BNH   SETRET              No..
         CSLGETP PLIST=(R2),     Get divisor address       X
               PARM=DIVISOR,                               X
               ADDRESS=(R6)
         L     R6,0(,R6)         Get divisor
         SR    R4,R4             Divide result by
         DR    R4,R6             divisor
         SPACE 1
SETRET   DS    0H
         CSLGETP PLIST=(R2),     Get result address        X
               PARM=RESULT,                                X
               ADDRESS=(R6)
         ST    R5,0(,R6)         Return result to caller
         SPACE 2

************************************************************
*  Return to caller
************************************************************
         SPACE 1
         SR    R15,R15                RC = 0
         CSLEXIT RETURN=(R15)         Return to caller
         EJECT
************************************************************
*  Constants and Literals
************************************************************
         SPACE 1
         LTORG *
         END   CALC
```

Template File for CALC Routine Parameters

The template file describing CALC's parameters is called CALC TEMPLATE, and it looks like this:

```
5  4
SBIN    4   OUTPUT     Return code
SBIN    4   OUTPUT     Result
SBIN    4   INPUT      Operand 1
SBIN    4   INPUT      Operand 2
SBIN    4   INPUT      Divisor (optional)
```

Note the following about the template file:

1. The first line says that CALC has 5 possible parameters, but only four are required.
2. Every parameter is a signed binary integer and is four bytes long.
3. The return code and result fields are outputs, operand 1, operand 2, and the divisor are inputs.
4. Comments are noted in the fourth column.

TBSUM: Example CSL Assembler Routine #2

The second example CSL routine is called TBSUM. It computes the sum of 2 or more columns in a table and stores the sum for each row in a separate column. It then calls another CSL routine called TBSORT so that the rows can be sorted using a character string column in the same table

TBSUM will be a direct call routine so the CSECT name of the ASSEMBLE routine cannot be the same as the CSL routine name. The CSECT name of TBSUMA has been chosen here.

TBSUMA ASSEMBLE

```
*************************************************************
*  Routine name :
*                TBSUMA
*
*  FUNCTION:
*          Computes the sum of the column entries for each
*          row of an input table. The sum of the column
*          elements for each row is stored in a corresponding
*          summation column element.
*
*           TBSUMA returns;
*                 -10 when too many parameters are passed
*                 -11 when too few parameters are passed
*                  8  when the number of table rows supplied
*                     is zero, negative or more than 1000
*    Parameters ;
*          RETURN - return code
*          ROWS   - number of rows in table
*          INDX   - column of row indices. Each element
*                   has the order of the row in a sorted list
*          ID     - 8 character row identification string
*          SUM    - holds the sum of each row's numeric
*                   columns
*          COL1-COL6  - numeric columns which are to be
*                       summed
*

*************************************************************
*   Register Equates
*************************************************************
    REGEQU
    EJECT
*************************************************************
*   Module entry logic
*************************************************************
TBSUMA AMODE 31
TBSUMA RMODE ANY
TBSUMA CSLENTRY DIRECT(7,4),(RETURN,ROWS,INDX,ID,SUM,      X
                COL1,COL2,COL3,COL4,COL5,COL6)
       LR    R12,R15                   Save base address and
       USING TBSUMA,R12                set up a new base register
       DROP  R15
*
       LR    R7,R1                     Save plist address
       LR    R3,R0                     Save number of
                                       parameters
*
       LA    R0,WORKLNTH               length of workarea
       CMSSTOR OBTAIN,BYTES=(0)        obtain storage
       LR    R15,R13                   establish
                                       addressability
       LR    R13,R1                    to work area
       USING WORKAREA,R13
*
       ST    R15,SAVEAREA+4            setup and chain
                                       saveareas
       ST    R13,8(,R15)
*

*************************************************************
*   Main program function
*************************************************************
     CSLGETP PLIST=(R7),PARM=ROWS,                         X
          ADDRESS=(R4)
     L     R4,0(R4)              get number of rows in table
     C     R4,=F'1000'           if number of rows > 1000
     BH    ERROR
     C     R4,=F'0'              or number of rows < 1
     BNH   ERROR                 then return with error code 8
*
     LR    R6,R4                 convert number of rows to the
     BCTR  R6,R0                 displacement of the last row
     SLL   R6,R2                 element
*
```

```
     S     R3,=F'5'             compute the number of columns
*                                   passed
*
*    Compute sums for each row of table and store sum in SUM
*    column
*

LOOPSTRT DS    0H
     SR    R8,R8                Zero summation register
*
     CSLGETP PLIST=(R7),PARM=COL1,  get address of COL1     X
           ADDRESS=(R9)
     A     R8,0(R6,R9)          add COL1 to summation
*
     CSLGETP PLIST=(R7),PARM=COL2,  get address of COL2     X
           ADDRESS=(R9)
     A     R8,0(R6,R9)          add COL2 to summation
*
*    The remaining columns are optional and therefore their
*    existence is verified before the row element of each
*    is added in.
*
     C     R3,=F'3'             was COL3 supplied
     BL    LOOPEND              if not then summation is
*                                   complete for this row
     CSLGETP PLIST=(R7),PARM=COL3,  get address of COL3     X
           ADDRESS=(9)
     A     R8,0(R6,R9)          add COL3 to summation
*
     C     R3,=F'4'             was COL4 supplied ?
     BL    LOOPEND              if not then summation is

*                                   complete for this row
     CSLGETP PLIST=(R7),PARM=COL4,  get address of COL4     X
           ADDRESS=(R9)
     A     R8,0(R6,R9)          add COL4 to summation
*
     C     R3,=F'5'             was COL5 supplied ?
     BL    LOOPEND              if not then summation is
*                                   complete for this row
     CSLGETP PLIST=(R7),PARM=COL5,  get address of COL5     X
           ADDRESS=(R9)
     A     R8,0(R6,R9)          add COL5 to summation
*
     C     R3,=F'6'             was COL6 supplied ?
     BL    LOOPEND              if not then summation is
*                               complete for this row
     CSLGETP PLIST=(R7),PARM=COL6,  get address of COL6     X
           ADDRESS=(R9)
     A     R8,0(R6,R9)          add COL6 to summation
*

LOOPEND  DS    0H
*
     CSLGETP PLIST=(R7),PARM=SUM,   get address of SUM      X
           ADDRESS=(R9)
     ST    R8,0(R6,R9)          store sum in summation entry
*                               for row
*
     S     R6,=F'4'             increment displacement to
*                                 next row
     LTR   R6,R6              processed all rows?
     BNM   LOOPSTRT       if not the continue with next
                                 row
*
*   Setup parameter list for call to TBSORT
*
     CSLGETP PLIST=(R7),PARM=RETURN,ADDRESS=(R3)
     CSLGETP PLIST=(R7),PARM=ROWS,ADDRESS=(R4)
     CSLGETP PLIST=(R7),PARM=INDX,ADDRESS=(R5)
     CSLGETP PLIST=(R7),PARM=ID,ADDRESS=(R6)
*
     CALL  TBSORT,((R3),(R4),(R5),(R6)),VL,MF=(E,TBSORTPL)
     LR    R6,R15                save return code
     B     RETURN
*
ERROR    DS     0H
         L      R6,=F'8'

************************************************************
*   Module exit logic
************************************************************
```

```
RETURN   LR    5,13
     L     13,4(13)
     LA    R0,WORKLNTH              length of workarea
     CMSSTOR RELEASE,BYTES=(0),ADDR=(5)
     CSLEXIT RETURN=(R6)            Return to caller
*
     LTORG
************************************************************
*  WORKING STORAGE
************************************************************
         SPACE
WORKAREA DSECT
SAVEAREA DS    18F
TBSORTPL CALL  ,(RET,ROW,INDX,STRING),MF=L
RET      DS    A
ROW      DS    A
STRING   DS    A
INDX     DS    A
WORKLNTH EQU   *-WORKAREA
         SPACE
         END   TBSUMA
```

The sorting routine, TBSORT in this example, is a generic name for a family of CSL routines. Each routine uses the same parameter list, but each is different in the sorting method it uses and the result it produces. All of the routines leave the result of their sort in the index column of the table. The entry in the index column relates to the sorted order of the rows. For example, the following list of numbers would have the corresponding index column if it was run through a routine which sorted it in ascending order:

```
         List        Index Column
          12               3
           7               1
           9               2
          14               4
          18               5
```

TBSORT1 is an example of one of the TBSUM routine family.

TBSORT1 ASSEMBLE

```
*******************************************************
*  Routine name :
*                TBSORT1
*
*  FUNCTION:
*     Sorts a table by the row number.
*
*     TBSORT1 returns;
*          -10 when too many parameters are passed
*          -11 when too few parameters are passed
*           16 when the number of table rows supplied is
*              zero, negative or more than 1000
*      Parameters ;
*           RETURN - return code
*           ROWS   - number of rows in table
*           INDX   - column of row indices. Each element
*                    has the order of the row in a
*                    sorted list
*           STR    - 8 character string  (not used)
*
**********************************************************
*   Register Equates
**********************************************************
    REGEQU
    EJECT
**********************************************************
*   Module entry logic
**********************************************************
TBSORT1  CSLENTRY DIRECT(4,0),(RETURN,ROWS,INDX,STR)
         LR    R12,R15           Save base address and
         USING TBSORT1,R12       set up a new base
                                 register
*
         LR    R7,R1             Save plist address
*

**********************************************************
```

```
*   Main program function
*************************************************************
  CSLGETP PLIST=(R7),PARM=ROWS,  get address of ROWS       X
        ADDRESS=(R4)
  L     R4,0(R4)                 get number of rows
                                 in table
  C     R4,=F'10000'             if number of rows >
                                 10000
  BH    ERROR
  C     R4,=F'0'                 or number of rows
                                 < 1
  BNH   ERROR                    then return with error
                                 code 16
*
*
CSLGETP PLIST=(R7),PARM=INDX,   get index row address     X
        ADDRESS=(R3)
*
  LA    R5,1                    initialize index
                                (row number)
LOOP     ST    R5,0(R3)            store index in index
                                   column
         LA    R5,1(R5)            increment index value
         LA    R3,4(R3)            increment to next row
         BCT   R4,LOOP             test for end of column
         SR    R15,R15             set return code of 0
         B     RETURN              return to caller
*
ERROR    DS    0H
         L     R15,=F'16'          set return code for
*                                  invalid table size

***********************************************************
*   Module exit logic
***********************************************************
RETURN   CSLEXIT RETURN(R15)          Return to caller
*
         LTORG
         END   TBSORT1
```

Template Files for TBSUM and TBSORT Routine Parameters

The template file describing TBSUM's parameters is called TBSUM TEMPLATE, and it looks like this:

```
DIRECT 11 7
SBIN    4    OUTPUT      Return code
TABLE   1000 INOUT       Maximum number of rows is 1000
 LEN    4    INPUT       Current number of rows in table
 C.UBIN 4    OUTPUT      Table index column for sorting
 C.CHAR 8    INPUT       Character Identification string
 C.SBIN 4    OUTPUT      Sum of numeric column entries
 C.SBIN 4    INPUT       numeric column 1
 C.SBIN 4    INPUT       numeric column 2
 C.SBIN 4    INPUT       numeric column 3 (optional)
 C.SBIN 4    INPUT       numeric column 4 (optional)
 C.SBIN 4    INPUT       numeric column 5 (optional)
 C.SBIN 4    INPUT       numeric column 6 (optional)
```

Note the following about the template file:

1. The first line says that TBSUM has 11 possible parameters, but only 7 are required.
2. The TABLE entry is not a parameter but is used to denote the start of the table structure.
3. The LEN entry following the TABLE entry specifies the number of rows actually existing in the table at the time of the call.
4. Some of the columns in the table are output directed while others are input directed.
5. Comments are noted in the fourth column.

The template file describing TBSORT's parameters is called TBSORT TEMPLATE, and it looks like this:

```
DIRECT  4 4
SBIN    4    OUTPUT      Return code
TABLE   1000 INOUT       Maximum number of rows is 1000
 LEN    4    INPUT       Current number of rows in table
 C.UBIN 4    OUTPUT      Table index column for sorting
 C.CHAR 8    INPUT       Character Identification string
```

Note the following about the template file:

1. The first line says that TBSORT expects all 4 parameters.
2. The DIRECT keyword on the first line signifies that the CSL routine using this template file is directly callable.

Control File for Building the Library

The control file showing the routines to go in the library is called MYLIB CSLCNTRL. When working with directly callable routines such as TBSUM and TBSORT the control file can be written in a number of different ways. This is one example.

```
*
ROUTINE TBSUM   TBSUMA  TBSUM   TEMPLATE A
        (Path 513.1  Subgroup TBSUM
*
ALIAS   TBSORT  Path 513.4
*
ROUTINE TBSORT1 TBSORT1 TBSORT  TEMPLATE A (Subgroup TBSUM
ROUTINE TBSORT2 TBSORT2 TBSORT  TEMPLATE A (Subgroup TBSUM
ROUTINE TBSORT3 TBSORT3 TBSORT  TEMPLATE A (Subgroup TBSUM
ROUTINE TBSORT4 TBSORT4 TBSORT  TEMPLATE A (Subgroup TBSUM
ROUTINE TBSORT5 TBSORT5 TBSORT  TEMPLATE A (Subgroup TBSUM
*
ROUTINE RTN1    CALC    CALC    TEMPLATE A (Subgroup TBSUM
*
```

The CSLGEN control file is MYLIB CSLCNTRL. The name TBSORT has no text deck associated with it, it is merely a direct call alias name. One of the five sort routines (TBSORT1 thru TBSORT5) will be assigned to that alias name at RTNLOAD time. Assume that text files TBSORT2 through TBSORT5 exist and represent other directly callable CSL routines in the TBSORT family.

Note the following about the control file:

1. TBSORT has a path assigned. It must have a path assigned because TBSUM makes a direct call to TBSORT.
2. Both TBSUM and TBSORT have been assigned unique paths. They could just as easily have been shared paths.
3. All of the routines have been assigned a subgroup name of TBSUM. This will make listing the routines by CSLMAP easier. The routines can also be loaded using the subgroup option.

Command to Build the Library

Assemble the CSL routines to get TEXT files. Then issue the following command to build a library called MYLIB CSLLIB, formatted for DASD, using the specified control file.

```
CSLGEN DASD MYLIB FROM MYLIB CSLCNTRL
```

Commands to Make the Library Accessible

The first command specifies that MYLIB should be searched first, followed by VMLIB:

```
GLOBAL CSLLIB MYLIB
```

```
RTNLOAD rtn1 tbsum
RTNLOAD tbsort1 tbsort (alias
```

will properly load all of the CSL routines in MYLIB to execute example #2, TBSUM. The subgroup option, specified in the CSLCNTRL file, lets you easily find your routines after they are loaded.

```
RTNLOAD rtn1 tbsum tbsort1 tbsort (alias
```

is a one-line alternative to the previous example.

```
CSLMAP * (group TBSUM )
```

will locate routines TBSUM and TBSORT after loading.

APPL1: Application Program #1

The following example VS FORTRAN program, called APPL1, invokes the *CALC* CSL routine several times. The *CALC* routine was loaded in MYLIB with the name *RTN1*.

```
C Sample VS FORTRAN program to invoke the 'CALC' CSL routine
        PROGRAM APPL1
C Declare DMSCSL as external file
        EXTERNAL DMSCSL
        EXTERNAL CALC
C Declare variables
        CHARACTER*8    ROUTIN
        INTEGER        RC
        INTEGER        TOTAL
        INTEGER        N1, N2, N3
C Initialize Variables
        ROUTIN  = 'RTN1    '
        N1      = 1
        N2      = 2
        N3      = 3
C Call the routine with no operands and display
  return code parameter
C  upon return. Use the DMSCSL interface.
        CALL DMSCSL(ROUTIN,RC)
        WRITE (6,30) 'RC    = ', RC
C Call the routine directly with one operand and
  display return code and total
C  parameters upon return.
        CALL CALC(RC,TOTAL,N1)
        WRITE (6,30) 'RC    = ', RC
        WRITE (6,30) 'TOTAL = ', TOTAL
C Call the routine directly with three operands and
  display return code and total
C parameters upon return.
        CALL CALC(RC,TOTAL,N1,N2,N3)
        WRITE (6,30) 'RC    = ', RC
        WRITE (6,30) 'TOTAL = ', TOTAL
  30    FORMAT (A9,I4)
        END
```

Figure 47. A FORTRAN Program Called APPL1

APPL2: Example Application Program #2

The following example VS FORTRAN program, called APPL2, invokes the *TBSUM* CSL routine. The *TBSUM* routine was loaded in MYLIB with the name *TBSUM*.

```
C Sample VS FORTRAN program to invoke the 'TBSUM' CSL routine
       PROGRAM APPL2
*
C Declare TBSUM as external file
       EXTERNAL TBSUM
*
C Declare variables
       INTEGER      RC
       INTEGER      LENGTH
       CHARACTER    STRING(5)*8
       INTEGER      ROWSUM(5)
       INTEGER      INDEX(5)
       INTEGER      COL1(5),COL2(5),COL3(5)
*
C Initialize variables
       DO 10 J = 1,5
       INDEX(J) = J
       ROWSUM(J) = 0
       COL1(J) = J
       COL2(J) = J
       COL3(J) = J
10     CONTINUE
       STRING(1) = 'E'
       STRING(2) = 'D'
       STRING(3) = 'C'
       STRING(4) = 'B'
       STRING(5) = 'A'
       LENGTH = 5
*
C Call the routine
       CALL TBSUM(RC,LENGTH,INDEX,STRING,ROWSUM,COL1,COL2,COL3)
       DO 20 J=1,5
       I = INDEX(J)
       PRINT *,INDEX(I), STRING(I), ROWSUM(I), COL1(I),
       COL2(I), COL3(I)
20     CONTINUE
       END
```

Figure 48. A FORTRAN Program Called APPL2

Here are sample runs of the two programs, APPL1 and APPL2:

```
Ready;
rtnload rtn1 (from mylib
Ready;
global csllib mylib
Ready;
load appl1 (start
Execution begins…
RC    =  -11
RC    =  -11
TOTAL =    0
RC    =    0
TOTAL =    1
Ready;
```

Figure 49. Sample Run of APPL1

```
Ready;
* the second example…
*
global txtlib vsf2fort mylib
RTNLOAD rtn1 tbsum
RTNLOAD tbsort1 tbsort (alias
global txtlib mylib
load appl2 (start
Execution begins…
 1 E        3 1 1 1
 2 D        6 2 2 2
 3 C        9 3 3 3
 4 B        12 4 4 4
 5 A        15 5 5 5
Ready;
```

Figure 50. Sample Run of APPL2

Chapter 18. Using Auxiliary Directories

This chapter describes how to:

- Add an auxiliary directory to CMS
- Create an auxiliary directory.

Overview of an Auxiliary Directory

When a disk is accessed, each module that fits the description specified on the ACCESS command is included in the resident directory. An **auxiliary directory** is an extension of the resident directory and contains the name and location of certain CMS modules that are not included in the resident directory. These modules, if added to the resident directory, would significantly increase its size, thus increasing storage requirements. An auxiliary directory can reference modules that reside on the S-disk; or, if the proper linkage is provided, reference modules that reside on any other read-only CMS disk. This chapter discusses how to add and create auxiliary directories.

To take advantage of the saving in storage, modules that are referenced with an auxiliary directory should never be in the resident directory. The disk where these modules reside should be accessed in a way that excludes these modules.

Adding an Auxiliary Directory

To add an auxiliary directory to a CMS minidisk, the system programmer must generate the directory, initialize it, and establish the proper linkage. Only when all three tasks are completed, can a module described in an auxiliary directory be properly located.

Generating the Auxiliary Directory

An auxiliary directory TEXT deck is generated by assembling a set of DMSFST macros, one for each module name. For more information on the DMSFST macro, see *z/VM: CMS Macros and Functions Reference*.

Initializing the Auxiliary Directory

After the auxiliary directory is generated with the DMSFST macro, it must be initialized. The CMS GENDIRT command initializes the auxiliary directory with the name and location of the modules to reside in an auxiliary directory. By using the GENDIRT command, the file entries for a given module are loaded only when the module is invoked.

Note: Do not load the modules into the transient area before issuing the GENDIRT command.

For more information on the GENDIRT command, see *z/VM: CMS Commands and Utilities Reference*.

Establishing the Proper Linkage

The CMS function, DMSLADAD, must be called by a user program or interface to initialize the directory search order. The subroutine, DMSLADAD, can be called with an SVC 202 or CMSCALL, with register 1 pointing to the appropriate PLIST. The disk containing the modules listed in the auxiliary directory must be accessed as the mode specified, or implied, by the GENDIRT command before the call is issued. If the GENDIRT command has not been used, the user receives the message: *File not found* or *Error reading file*.

The coding necessary for the call is:

```
         CMSCALL PLIST=PLIST,MODIFY=YES
```

This call must be executed before the call to any module to be located with an auxiliary directory.

The PLIST should be:

```
PLIST  DS   0F
       DC   CL8'DMSLADAD'
       DC   V(directoryname)
       DC   F'0'
       DC   8X'FF'
```

The address pointing to `directoryname` has to be below 16MB, because the PLIST will be using 24-bit addressing.

The auxiliary directory is copied into nucleus free storage. The directory information for the target mode expressed or implied by the GENDIRT command is found and its file directory address chain is modified to include the nucleus copy of the auxiliary directory.

The address of the nucleus copy of the auxiliary directory is saved in the third parameter of the input PLIST and the high-order byte of the third parameter is set to X'80' to indicate that the directory search chain was modified and that the next call to DMSLADAD is a clear request.

To reset the directory search chain, a second call is made to DMSLADAD using the modified PLIST. DMSLADAD removes the nucleus copy of the auxiliary directory from the chain and frees it. This call to DMSLADAD removes all auxiliary blocks from the directory chain; there is no linkage to delete selective auxiliary directory blocks from the chain. DMSLADAD does not, however, restore the caller's PLIST to its initial state.

Error Handling and Return Codes

An error handling routine should be coded to handle nonzero return codes from DMSLADAD in register 15. The following errors (with condition code = 2) may occur on a call to DMSLADAD:

Description	DMSLADAD Request Type	Return Code
The auxiliary directory address is not specified in the PLIST.	initialize, clear	1
The target mode specified at GENDIRT time is not accessed.	initialize	1
The target mode specified at GENDIRT time accesses an OS or DOS disk.	initialize	1
The address of the nucleus copy of the auxiliary directory is not specified in the third parameter of the PLIST.	clear	2
No auxiliary directory has been initialized on the given disk.	clear	2
The target mode specified at GENDIRT time is accessed in shared storage.	initialize	3

Creating an Auxiliary Directory

In this example, consider an application called PAYROLL consisting of several modules. It is possible to put these modules in an auxiliary directory rather than in the resident directory. It is further possible to put the auxiliary directory on a disk other than the system disk. In this example, the auxiliary directory is placed on the Y-disk.

First, generate the auxiliary directory TEXT deck for the payroll application using the DMSFST macro:

```
PAYDIRT  START   0
         DC      F'40'  LENGTH OF FST ENTRY:fnref refid=forme.
         DC      A(DIRTEND-DIRTBEG) SIZE OF DIRECTORY
DIRTBEG  EQU     *
         DMSFST  PAYROLL1
         DMSFST  PAYROLL2
         DMSFST  PAYROLL3
         DMSFST  PAYFICA
         DMSFST  PAYFEDTX
         DMSFST  PAYSTATE
         DMSFST  PAYCITY
         DMSFST  PAYCREDU
         DMSFST  PAYOVERT
         DMSFST  PAYSICK
         DMSFST  PAYSHIFT
         DC      2A(0) POINTER TO NEXT FST BLOCK
DIRTEND  EQU     *
         END
```

In this example, the payroll control program (PAYROLL), the payroll auxiliary directory (PAYDIRT), and all the payroll modules reside on the 194 disk.

In the payroll control module (PAYROLL), the subroutine DMSLADAD must be called to establish the linkage to the auxiliary directory. This call must be executed before any call is made to a payroll module that is in the PAYDIRT auxiliary directory.

```
          LA   R1, PLIST
          SVC  202
          DC   AL4(ERRTN)

PLIST     DS   0F
          DC   CL8'DMSLADAD'
          DC   V(PAYDIRT)
          DC   F'0'
```

Next, all payroll modules must have their absolute core-image files generated and the payroll auxiliary directory must be initialized. In the example, the payroll control module (PAYROLL) is given a mode number of 2 while the other payroll modules are given a mode number of 1. When the PAYROLL program is finally executed, only the files on the 194 disk with a mode number of 2 are accessed. This means only the PAYROLL control program (which includes the payroll auxiliary directory) will be referenced from the resident directory. All the other payroll modules, because they have mode numbers of 1, are referenced with the payroll auxiliary directory.

The following sequence of commands create the absolute core-image files for the payroll modules and initialize the payroll auxiliary directory.

```
ACCESS 194 A
LOAD PAYROLL PAYDIRT
GENMOD PAYROLL       (now the auxiliary directory is included
                     in the payroll control module, but it is
                     not yet initialized.)
LOADMOD PAYROLL
INCLUDE PAYROLL1
GENMOD PAYROLL1      (this sequence of three commands is
   .                 repeated for each payroll module called
   .                 by PAYROLL to establish the proper
   .                 address where the module would be loaded.)
LOADMOD PAYROLL
INCLUDE PAYSHIFT
GENMOD PAYSHIFT

LOADMOD PAYROLL
GENDIRT PAYDIRT Y
GENMOD PAYROLL MODULE A2
```

When it is time to execute the PAYROLL program, the 194 disk must be accessed as the Y-disk (the same mode letter as specified on the GENDIRT command). Also, the 194 disk is accessed in a way that includes

[12] F‘64’ should be used if FORM=E is specified on DMSFST macro.

the PAYROLL control program in the resident directory but not the other payroll modules. This is done by specifying a mode number of 2 on the ACCESS command.

```
ACCESS 194 Y/S * * Y2
```

Now, a request for a payroll module, such as PAYOVERT, can be successfully fulfilled. The auxiliary directory will be searched and PAYOVERT will be found on the Y-disk.

Note:

1. A disk referred to by an auxiliary directory must be accessed as a read-only disk, and it cannot be accessed in shared storage. (For details on the ACCESS command, see the *z/VM: CMS Commands and Utilities Reference*.)
2. You cannot issue the GENDIRT command against an auxiliary directory included in a transient module, since the GENDIRT command is also a transient module. In the example above, if you issue:

   ```
   ACCESS 194 A
   LOAD PAYROLL PAYDIRT (ORIGIN TRANS
   ⋮
   LOADMOD PAYROLL
   GENDIRT PAYDIRT Y
   ```

   the GENDIRT module overlays the PAYROLL module in the transient area before initializing the PAYDIRT auxiliary directory, and hence, would fail.

Part 4. Connectivity Programming in CMS

Connectivity is the ability of one program to communicate with another program. System Network Architecture (SNA) defines various sets of rules for data to be transmitted in a network. Application programs communicate with each other using a layer of SNA called Advanced Program-to-Program Communication (APPC). APPC is also known as LU 6.2. VM implements the base set of APPC with APPC/VM.

Part 4, " Connectivity Programming in CMS," on page 277 includes the following chapters:

- Chapter 19, "CMS Support of IUCV," on page 279 describes IUCV connectivity between virtual machines and between virtual machine and CP system services.
- Chapter 20, "APPC/VM Assembler Interface," on page 287 describes the assembler programming interface for APPC/VM, which allows communications between application programs that are written in assembler language.
- Chapter 21, "Using Advanced APPC/VM Functions," on page 307 describes more advanced APPC/VM functions that you can use in your programs.

Chapter 19. CMS Support of IUCV

This chapter describes the CMS support for the Inter-User Communications Vehicle (IUCV). IUCV is a communications facility that enables a program running in a virtual machine to communicate with programs in other virtual machines, with a CP system service, and with itself. Before continuing with this chapter, you should become familiar with the concepts and information about IUCV as described in *z/VM: CP Programming Services*. That book contains complete information on IUCV in z/VM. Assuming you are now familiar with IUCV, let us continue.

The CMS support for IUCV makes it easier for multiple programs operating within the same virtual machine to use IUCV functions without interfering with each others' use of IUCV. Because CMS manages IUCV interrupts in a virtual machine, CMS will route IUCV interrupts to the appropriately defined IUCV interrupt handler exit routines. IUCV interrupt handler exit routines are defined by programs to CMS and are identified by a ***name***. Each ***name*** must be unique within a virtual machine, therefore, allowing programs to handle their *own* IUCV interrupts and paths; also keeping other programs from IUCV interrupts and paths that they don't *own*.

In CMS, you can invoke IUCV functions through the CMS macros called HNDIUCV and CMSIUCV. The HNDIUCV macro provides the functions necessary to establish, change and end a program's IUCV environment. The CMSIUCV macro provides the functions necessary to start or end IUCV communications with a partner. The following table summarizes the functions provided by the HNDIUCV and CMSIUCV macros which are useful for IUCV. For detailed information about these functions see *z/VM: CMS Macros and Functions Reference*.

Macro	Function	Description
CMSIUCV	ACCEPT	Accept a connection request from another virtual machine in the same system.
CMSIUCV	CONNECT	Begin communications with another virtual machine in the same system or with a CP system service.
CMSIUCV	SEVER	Terminate communications with another virtual machine in the same system or with a CP system service.
HNDIUCV	CLR	Terminate a program's IUCV environment
HNDIUCV	REP	Replace currently-defined fields for a particular IUCV program in CMS.
HNDIUCV	SET	Initialize a program's IUCV environment

Note that the CMSIUCV and HNDIUCV macros do not provide interfaces to all of the IUCV capability provided in CP. Therefore, the CP IUCV macro must be executed to CP to get this function. Note also that the CMSIUCV and HNDIUCV macros provide functions intended to be used only by the CMS support of Advanced Program-to-Program Communications/VM (APPC/VM). The CMSIUCV macro functions, QCMSWID and RESOLVE, are intended only for APPC/VM. Although the HNDIUCV functions, HLD (HOLD) and RES (RESUME) are intended to control the handling of APPC/VM private resource connection pending interrupts, they also affect the handling of IUCV interrupts. An HNDIUCV SET name *on hold* will cause any IUCV connection pending interrupts to be immediately severed by CMS. HNDIUCV HLD puts a name *on hold* and HNDIUCV RES clears the *on hold* condition for a name. For more detail about using APPC/VM in CMS refer to Chapter 20, "APPC/VM Assembler Interface," on page 287. APPC/VM enables a program to communicate with a resource manager program that resides in the same z/VM system, within the same TSAF or CS collection, or anywhere within a network defined by IBM's System Network Architecture (SNA) for APPC.

Now that we know what CMS macros are provided by the CMS support for IUCV, let us describe how to use them. First, your program declares to CMS its intent to use IUCV by using the HNDIUCV SET macro function. The EXIT= parameter on HNDIUCV SET identifies the interrupt handler exit routine which should get control when CMS gets IUCV connection pending interrupt for the ***name*** specified on the NAME= parameter. Other programs will use ***name*** as the value for the USERDTA= field on IUCV CONNECT when establishing an IUCV path with your program.

After successfully calling HNDIUCV SET, your program is ready to both receive IUCV connection pending interrupts for the specified ***name*** and to initiate IUCV communication with other programs. Let us examine both cases.

- To start IUCV communications with another program, your program would first build an IUCV CONNECT parameter list with the MF=L option and then issue CMSIUCV CONNECT, passing CMS the IUCV CONNECT parameter list. Your call to CMSIUCV CONNECT should specify on the NAME= parameter the same ***name*** as on HNDIUCV SET. Note also that there is an EXIT= parameter on CMSIUCV CONNECT; this allows you to specify a different interrupt handler exit routine from the one that was specified on HNDIUCV SET.

  If your partner accepts your connection, your interrupt handler exit routine will be driven with a connection complete interrupt. Now, you can use the IUCV SEND, RECEIVE or other functions provided by the CP IUCV macro; and your interrupt handler exit routine will be driven for all interrupts associated with this IUCV path.
- If another program can communicate with your program, the interrupt handler exit routine specified on HNDIUCV SET will get control for a connection pending interrupt. Your program can accept the path by building an IUCV ACCEPT parameter list and issuing CMSIUCV ACCEPT. Your call to CMSIUCV ACCEPT should specify on the NAME= parameter the same ***name*** as on HNDIUCV SET. Note also that there is an EXIT= parameter on CMSIUCV ACCEPT; like CMSIUCV CONNECT, this will allow you to specify a different interrupt handler exit routine from the one that was specified on HNDIUCV SET.

  If your CMSIUCV ACCEPT function completes indicating success, an IUCV path is established and you can use the IUCV SEND, RECEIVE or other functions provided by the CP IUCV macro; and your interrupt handler exit will be driven for all interrupts associated with this IUCV path.

When your program is ready to end IUCV communications on a path or your partner has terminated IUCV communications with your program, set up an IUCV SEVER parameter list and issue CMSIUCV CONNECT. Finally, when your program is all done using IUCV, use the HNDIUCV CLR function to have CMS discontinue the tracking of your ***name*** and the interrupt handler exit routine associated with the ***name***.

The next section shows an example using the CMS support for IUCV.

Using IUCV in CMS to Communicate Between Two Virtual Machines

Figure 51 on page 281 shows an example of how HNDIUCV and CMSIUCV macro instructions can be issued by two virtual machines communicating with each other in CMS.

```
 Virtual Machine X - Source           Virtual Machine Y - Target

 1. HNDIUCV SET,NAME=ONE,EXIT=A        1. HNDIUCV SET,NAME=TWO,EXIT=X1
 2. Set up the IUCV CONNECT
     parameter list
 3. CMSIUCV CONNECT,NAME=ONE,EXIT=B
                                       4. Connection pending external
                                          interrupt
                                       5. EXIT X1 receives control
                                       6. Set up the IUCV ACCEPT
                                           parameter list
                                       7. CMSIUCV ACCEPT,NAME=TWO,EXIT=X2
 8. Connect-complete external
    interrupt
 9. EXIT B receives control
    .
    .                                     .
10. Programs in virtual machines X and Y communicate using
        IUCV functions such as SEND and RECEIVE.
    .                                     .
    .                                     .
11. Set up the IUCV SEVER
     parameter list
12. CMSIUCV SEVER,NAME=ONE                .
                                      13. SEVER external interrupt
                                      14. EXIT X2 receives control
                                      15. Set up the IUCV SEVER
                                           parameter list
                                      16. CMSIUCV SEVER,NAME=TWO
17. HNDIUCV CLR,NAME=ONE              17. HNDIUCV CLR,NAME=TWO
    .                                     .
    .                                     .
    .                                     .
18. ONE   DC   CL8'RED'
                                      19. TWO   DC   CL8'BLUE'
20. A      exit routine               20. X1      exit routine
21. B      exit routine               21. X2      exit routine
```

Figure 51. Sequence of Instructions in Virtual Machine to Virtual Machine Communication

The following list is an explanation of the sequence of instructions used in Figure 51 on page 281.

1. A program running in virtual machine X wishes to communicate with a program running in virtual machine Y. Each program must independently issue the HNDIUCV macro to begin IUCV communications. By issuing HNDIUCV SET, CMS invokes the IUCV DCLBFR function and enables for IUCV external interrupts (bit 30 of control register 0) if the virtual machine's IUCV environment has not already been initialized. The EXIT parameter specifies the label of the exit routine to handle "connection pending" external interrupts for this HNDIUCV SET name. The name specified in the “NAME=name” parameter is required in all subsequent CMSIUCV macro functions for connection links (IUCV path IDs) associated with this name.
2. Before issuing a CMSIUCV CONNECT, the source program must set up an IUCV CONNECT parameter list (using MF=L). The IPVMID field of the IUCV CONNECT parameter list contains the user ID of the virtual machine you are connecting to (virtual machine Y). The first eight bytes of the IPUSER field of the IUCV CONNECT parameter list contain the eight-character identifying name of the program that issued a HNDIUCV SET in virtual machine Y. (This name must match the name specified on the HNDIUCV SET macro function issued by the program in virtual machine Y.) In this example, the first eight bytes of the IPUSER field equals ‘BLUE ’.
3. The program in virtual machine X issues a CMSIUCV CONNECT to initiate a communication link with virtual machine Y. By issuing CMSIUCV CONNECT, CMS invokes the IUCV CONNECT function. The ‘EXIT=B’ parameter causes CMS to associate the exit routine at label B with the IUCV path ID. (If the ‘EXIT=B’ parameter was omitted, CMS would associate the exit routine at label A – the one specified in the HNDIUCV SET macro – with the IUCV path ID.)
4. Virtual machine Y receives a connection-pending external interrupt as a result of the CMSIUCV CONNECT issued by the program in virtual machine X.
5. "EXIT X1" receives control as a result of the external interrupt. ("EXIT X1" receives control because it was specified on the EXIT parameter of the HNDIUCV SET macro, and the first eight bytes of

the IPUSER field in the connection pending external interrupt, match the field at the label (TWO) specified in the NAME parameter of the HNDIUCV SET macro.)

6. Before issuing a CMSIUCV ACCEPT, the target program must set up an IUCV ACCEPT parameter list (using MF=L). The IUCV path ID is the same as the one in the connection pending external interrupt.
7. To complete the connection, the program in virtual machine Y issues a CMSIUCV ACCEPT. By issuing CMSIUCV ACCEPT, CMS invokes the IUCV ACCEPT function. This completes the IUCV communication link with virtual machine X. The 'EXIT=X2' parameter also associates the exit routine at label X2 with the path ID. (If the 'EXIT=X2' parameter was omitted, CMS would associate the exit routine at label X1 – the one specified in the HNDIUCV SET macro – with the IUCV path ID.)
8. Virtual machine X receives a connection complete external interrupt as a result of the CMSIUCV ACCEPT issued by the program in virtual machine Y.
9. "EXIT B" receives control as a result of the external interrupt. ("EXIT B" receives control because it is specified on the EXIT parameter of the CMSIUCV CONNECT macro.)
10. The programs in virtual machine X and virtual machine Y carry on a conversation using IUCV macro functions. The most basic conversations use SEND and RECEIVE, but other IUCV functions can be used in a more advanced program.
11. Before issuing the CMSIUCV SEVER, the application in virtual machine X must set up a sever parameter list using IUCV SEVER with MF=L.
12. Virtual machine X completed its communications with virtual machine Y and terminates the IUCV communication link. The program in virtual machine X issues an CMSIUCV SEVER to terminate this link. By issuing CMSIUCV SEVER, CMS invokes the SEVER function and clears the exit associated with the IUCV path ID.
13. Virtual machine Y receives a SEVER external interrupt as a result of the CMSIUCV SEVER issued by virtual machine X.
14. "EXIT X2" receives control as a result of the external interrupt. ("EXIT X2" receives control because it was specified on the EXIT parameter of the CMSIUCV ACCEPT macro.)
15. Before issuing the CMSIUCV SEVER, the program in virtual machine Y must set up a sever parameter list using IUCV SEVER with MF=L.
16. The program then issues a CMSIUCV SEVER to terminate the communication link. By issuing CMSIUCV SEVER, CMS invokes the IUCV SEVER function and clears the exit associated with the communication link.
17. After all communications are complete and all communication paths have been severed, the program in virtual machine X and the program in virtual machine Y independently issue HNDIUCV CLR. HNDIUCV CLR terminates IUCV communications and clears the exit for connection pending interrupts for the name specified in the NAME parameter. CMS invokes the IUCV RTRVBFR function and disables IUCV external interrupts (bit 30 in control register 0) if there are no other programs in the virtual machine using IUCV.
18. This is the label specified in the NAME parameter. This location contains the identifying name of the program in virtual machine X. The name of this program is RED.
19. This is the label specified in the NAME parameter. This location contains the identifying name of the program in virtual machine Y. The name of this program is BLUE.
20. This is the label of an exit (interrupt handler) routine specified in the EXIT parameter of the HNDIUCV SET macro. This routine receives control when a connection pending interrupt occurs for the associated HNDIUCV SET name.
21. This is the label of an exit (interrupt handler) routine specified in the EXIT parameter of the CMSIUCV CONNECT or ACCEPT macros. This routine receives control when an interrupt occurs for the associated path ID.

Understanding Exit Routines

An exit routine receives control whenever an IUCV external interrupt occurs on an IUCV path. When the program's IUCV external interrupt routine is given control, all interrupts are disabled. The exit routine is responsible for providing proper entry and exit linkage for its IUCV external interrupt handling routine. The exit routine has the following requirements:

- The routine should not enable itself for any type of interrupts.
- The routine should not perform any I/O operations, since all interrupts are disabled.
- The routine must return control to the address in register 14.

When the routine receives control, the significant registers contain:

Table 26. Contents of registers

Register	Contents			
0	UWORD (user word) Field			
1	Points to a SAVEAREA in this format:			
	Label	Displacement		Contents
		Dec	Hex	
	GRS	0	0	General purpose registers 0–15 at the time of the interrupt.
	FRS	64	40	Floating-point registers 0–7 at the time of the interrupt.
	PSW	96	60	External Old PSW at the time of the interrupt.
	UREA	104	68	Register save area for exit routine's use.
	END	176	B0	End of save area
2	Address of the IUCV External Interrupt Buffer			
13	Points to the register save area at label UAREA for use by the exit routine			
14	Return address			
15	Entry point address			

Guidelines for Using the CMS Support of IUCV

Some IUCV macro functions affect the IUCV environment of the entire virtual machine. Because CMS cannot intercept any IUCV macro functions directly issued by a program, any CMS application program using IUCV has certain limitations on its use of IUCV functions. The program must not issue any IUCV function that interferes with the operation of other IUCV applications running in the same virtual machine.

Each IUCV macro function is listed below, along with an explanation of how the IUCV function can be used in a CMS application program. If a program does issue any IUCV functions listed as "Should not be used...", other programs using IUCV in CMS in the same virtual machine may be affected.

Use the following functions only to set up the parameter list:

ACCEPT
Is invoked by a program using CMSIUCV ACCEPT. It should not be issued directly by a program in CMS, except to set up an IUCV ACCEPT parameter list (MF=L).

CONNECT
Is invoked by a program using CMSIUCV CONNECT. It should not be issued directly by a program in CMS, except to set up an IUCV CONNECT parameter list (MF=L).

SEVER
Is invoked by a program using CMSIUCV SEVER. This function should not be issued directly by a program in CMS, except to set up an IUCV SEVER parameter list (MF=L). CMSIUCV SEVER checks to

make sure that the IPALL bit is not turned on in the IPFLAGS1 parameter list, because this would sever all paths in the virtual machine.

Use these functions freely, but with caution as noted under each function:

PURGE
Can be issued directly by a program in CMS.

QUERY
Can be issued directly by a program in CMS. This function is also used by HNDIUCV to determine the size of the external interrupt buffer and the maximum number of connections for this virtual machine.

QUIESCE
Can be issued directly by a program in CMS to quiesce a specific path. However, the issuing program must be careful that the IPALL bit is not turned on in the IPFLAGS1 parameter list, because this would quiesce all paths in the virtual machine.

RECEIVE
Can be issued directly by a program in CMS. However, the issuing program must be careful that a specific message ID or path ID is specified in the IUCV parameter list. If it is not, IUCV receives the first message that has not yet been partially received for the entire virtual machine, and this message may not belong to the program that issued the RECEIVE.

REJECT
Can be issued directly by a program in CMS.

REPLY
Can be issued directly by a program in CMS.

RESUME
Can be issued directly by a program in CMS to resume a specific path. However, the issuing program must be careful that the IPALL bit is not turned on in the IPFLAGS1 byte of the parameter list, because that would resume all paths in the virtual machine.

SEND
Can be issued directly by a program in CMS.

TESTCMPL
Can be issued directly by a program in CMS. However, the issuing program must be careful that the specified message ID or path ID is also specified in the IUCV parameter list. If it is not, IUCV completes the first message on the REPLY queue for the entire virtual machine, and this message may not belong to the program that issued the TESTCMPL.

Do not use these functions in a CMSIUCV program:

DCLBFR
Should not be issued directly by a program in CMS. This function is used by HNDIUCV SET to initialize the virtual machine's IUCV environment.

DESCRIBE
Should not be used by a program in CMS. This function clears the pending-message external interruption for the described message. This interrupt may not belong to the issuer of the DESCRIBE function; as a result, other programs running in the same virtual machine may be affected because the message is lost and never reflected to the true target.

RTRVBFR
Should not be issued directly by a program in CMS. The HNDIUCV CLR function and CMS abend processing use RTRVBFR to terminate the virtual machine's IUCV environment.

SETMASK
Should not be used by a program in CMS. This function disables certain IUCV external interrupts for the entire virtual machine; as a result, other programs running in the same virtual machine may be affected.

SETCMASK
Should not be used by a program in CMS. This function disables certain IUCV external interrupts for the entire virtual machine; as a result, other programs running in the same virtual machine may be affected.

TESTMSG
: Should not be used by a program in CMS. This function places the entire virtual machine in a wait state if no incoming messages or replies are pending; as a result, other programs running in the same virtual machine may be affected.

Chapter 20. APPC/VM Assembler Interface

In its simplest form, connectivity is the ability of one program to communicate with another program. In this book, we are concerned with communications between two application programs. Application programs are typically written to communicate with one another because a user needs access to some kind of data.

Systems Network Architecture (SNA) defines various sets of rules for data to be transmitted in a network. Application programs communicate with each other using a layer of SNA called **Advanced Program-to-Program Communication (APPC)**. APPC is also known as **LU 6.2**. VM implements the base set of APPC and several APPC option sets using **Advanced Program-to-Program Communication/VM (APPC/VM)**.

VM provides two programming interfaces to APPC/VM:

1. A low-level interface intended for programs written in assembler language.
2. **Common Programming Interface (CPI) Communications** (also known as SAA* communications interface).

   Note: Refer to the *Common Programming Interface Communications Reference (https://publibfp.dhe.ibm.com/epubs/pdf/c2643999.pdf)* for more details about CPI Communications routines.

The following communications programming interface is also available in z/VM:

- The Inter-User Communication Vehicle (IUCV), which is for communications between two programs on the same VM system. IUCV also allows a program to communicate with a CP system service. For complete information on IUCV, see the *z/VM: CP Programming Services*.

Overview of APPC/VM Assembler Interface

The assembler programming interface for APPC/VM allows communications between application programs that are written in assembler language. The APPC/VM assembler interface implements the base set of APPC (SNA LU 6.2) verbs and several APPC option sets.

The APPC/VM assembler interface provides macros and parameter lists that applications can use to set up and control the communications environment within one VM system, and among VM systems in a TSAF or CS collection. (For programs communicating outside a TSAF or CS collection to an SNA network, AVS translates APPC/VM protocols into APPC/VTAM*, which is the VTAM implementation of APPC.)

In addition, VM has implemented some IUCV functions for an APPC/VM environment. IUCV functions are not part of the APPC architecture and are unique to VM. For more information on how IUCV relates to APPC/VM, see "How APPC/VM Relates to General IUCV" on page 303.

This chapter gives an overview of the APPC/VM assembler interface, describes various APPC/VM assembler functions, shows how to write assembler APPC/VM programs in CMS, and discusses the relationship between APPC/VM and IUCV.

Note: The remainder of this book refers to the APPC/VM assembler programming interface as simply APPC/VM.

Basics of APPC/VM

The following sections describe some of the basics of APPC/VM communication: paths, states, and interrupts.

APPC/VM Paths

An APPC/VM path is a logical connection between one or more virtual machines. Information flows on APPC/VM paths. To establish an APPC/VM path between two virtual machines, at least one of the virtual machines must be authorized in the IUCV directory statement.

A path is created when the source virtual machine invokes the CONNECT function and the target virtual machine invokes the ACCEPT function. After the path is created, communications can begin. Programs identify a path using the PATHID parameter of the pertinent APPC/VM function.

The target virtual machine can prevent the path from being established by invoking the SEVER function. Either virtual machine can destroy an established path with the SEVER function.

A single virtual machine can have up to 65,536 APPC/VM paths defined. Two virtual machines can have more than one path between them. Communication can occur over multiple paths at the same time.

An APPC conversation is represented within a TSAF or CS collection as an APPC/VM path. A path exists for the life of the conversation. The APPCVM CONNECT is analogous to the APPC ALLOCATE because it creates a path; the APPCVM SEVER is analogous to the APPC DEALLOCATE because it destroys a path.

Note: SNA sessions have no representation in a TSAF or CS collection. VTAM allocates and ends the SNA sessions on which APPC/VM conversations through an SNA network are established.

APPC/VM States

The APPC/VM interface is a half-duplex communications protocol. This means that only one of the communications partners can send data at a time. Because of this, APPC/VM uses states to define what functions a program can and cannot issue at any given time.

A program is always in a single state for a particular conversation. When your program or your communications partner issues an APPC/VM function, the state of the conversation may change. If your virtual machine is communicating with different virtual machines through various paths, it may be in different states on different paths at the same time. A program participating in multiple conversations could have multiple states, too.

The basic states for APPC/VM assembler programs are:

Reset
The state for each program before communications begin and after communications end.

Connect
The state for a source program after a connection has been started but before it has completed, or the state for a target program after it has received a connection pending interrupt but before it has accepted.

Send
The state in which a program is allowed to send data.

Receive
The state in which a program is ready to receive data.

Confirm
The state in which a program must respond to its communications partner.

Sever
The state a program is in when its partner stops communications.

These APPC/VM states are based on the states that APPC defines, but there are two differences:

1. The Connect state is unique to APPC/VM
2. The Sever state is analogous to the APPC Deallocate state.

APPC/VM Interrupts

In APPC/VM, your program may receive notification of pending functions through external ***interrupts***. Interrupts are caused by actions taken by the virtual machine on the other end of the local APPC/VM path. Interrupts indicate pending and completed functions. For example, your virtual machine receives an interrupt when another virtual machine sends you a message that it wants you to receive.

At the start of your program, you should create a buffer to hold the interrupt information for an established APPC/VM path. (You can create this buffer using the HNDIUCV SET or IUCV DCLBFR function.) This buffer is a 40-byte area called an external interrupt buffer. There are two types of external interrupt buffers, control and application. For a complete description of these buffers, see the section on the IUCV DCLBFR (Declare Buffer) function in the *z/VM: CP Programming Services*. When your program is presented with an interrupt, information about the interrupt goes into one of these external interrupt buffers.

The possible APPC/VM interrupts you can get fall into two categories. The first type of interrupt signals that your communications partner has invoked some function, independent of your actions. These interrupts are:

- Connection pending
- Message pending
- Request-to-Send
- Sever.

The second type of interrupt signals the completion of a function that you initiated. These interrupts are:

- Connection complete
- Function complete.

The six basic types of APPC/VM interrupts are described in the following paragraphs.

Connection Pending External Interrupt

You get a connection pending interrupt when a virtual machine invokes an APPCVM CONNECT function to connect to your virtual machine. The interrupt is placed in a control buffer if the virtual machine wants to connect to a private resource; otherwise, the interrupt is presented to your virtual machine's application buffer.

Message Pending External Interrupt

You get a message pending interrupt when your communications partner issues an APPC/VM function for which you should issue an APPCVM RECEIVE. Any of the following APPCVM macro functions issued by your communications partner can cause a message pending interrupt:

- RECEIVE
- SENDCNF
- SENDDATA
- SENDERR.

You only get a message pending interrupt if you are in Receive state on the corresponding path.

Request-to-Send External Interrupt

You get a request-to-send interrupt when your communications partner issues the APPCVM SENDREQ function to request to send data.

Sever External Interrupt

You get a sever interrupt when the program to which you are connected or trying to connect to invokes an APPCVM SEVER or IUCV SEVER, invokes an HNDIUCV CLR (or IUCV RTRVBFR), or abends. You could also get a sever interrupt when the virtual machine to which you are connected or trying to connect to resets its virtual machine or logs off.

Note: If you get a sever interrupt after you have issued an APPC/VM function to your partner, do not assume that your function has terminated.

Connection Complete External Interrupt

You get a connection complete interrupt when you issue the APPCVM CONNECT WAIT=NO function and the virtual machine on the other end of the local APPC/VM path accepts the connection.

When you get a connection complete interrupt, do not assume that the target program performed any action to cause your function to complete.

Function Complete External Interrupt

You get a function complete interrupt (FCI) when the function that you issued completes. The completion of any of the following APPC/VM functions can cause a function complete interrupt:

- RECEIVE
- SENDCNF
- SENDDATA
- SENDERR
- SEVER.

For more information on APPC/VM external interrupts, see the *z/VM: CP Programming Services*.

Invoking APPC/VM Communication Functions

VM programs at each end of an APPC/VM path use APPC/VM functions to communicate with each other. Most APPC/VM communications functions are provided as parameters of the APPCVM macro. The APPC/VM communications functions used with the APPCVM macro are:

- CONNECT
- RECEIVE
- SENDCNF
- SENDCNFD
- SENDDATA
- SENDERR
- SENDREQ
- SEVER.

In addition, some APPC/VM functions are provided as parameters on the IUCV macro. The APPC/VM communications functions used with the IUCV macro are:

- ACCEPT
- QUERY
- SEVER.

The IUCV macro functions relate to both APPC/VM and IUCV paths. The IUCV functions are not defined by the SNA LU 6.2 (APPC architecture) verb interface, but they are a necessary complement for APPC programs executing in a VM processor. The IUCV macro functions that relate to APPC/VM are:

- ACCEPT
- CONNECT
- DCLBRF
- DESCRIBE
- QUERY
- RTRVBFR
- SETCMASK
- SETMASK
- SEVER
- TESTCMPL

- TESTMSG.

Note: Other IUCV macro functions are possible from APPC/VM, but are not recommended for use by APPC/VM programs running in CMS. See "How APPC/VM Relates to General IUCV" on page 303 for more information about these other IUCV functions.

Using Basic APPC/VM Functions

To write starter APPC/VM programs, you need to know the APPC/VM functions that do the basic steps of starting a conversation, communicating in a conversation, and ending a conversation.

Starting a Conversation

To start a conversation, your user program must issue an APPCVM CONNECT with a resource ID. Depending on the type of connection, a user program also might supply a ***connection parameter list extension***, which contains detailed information that is necessary to make a connection. (If the resource ID maps to an entry in a CMS communications directory file, the program generally does not have to build the extension itself, CMS does it.)

When your program issues an APPCVM CONNECT, your communications partner gets a connection pending interrupt. Your partner should examine the interrupt before accepting or rejecting the connection. The interrupt contains information such as the resource ID for which the connection is being made and the user ID of the requesting program.

In addition to the connection pending interrupt, your partner can also get other kinds of data before accepting the connection:

- *Allocate data*, which provides more details about the pending connection
- *Program Initialization Parameters (PIP data)*, which can serve many purposes. (See "Sending and Receiving Early Information" on page 308 for more information.)

After examining all this data, your communications partner can do either of the following in response to your connect request:

- Accept the connection if it wants to communicate with your program, making sure to specify the path ID that was on the connection pending interrupt.
- Accept the connection and then immediately sever the connection if it does not want to communicate with your program.

Sending and Receiving Data on the Conversation

When you issue the command to connect to a target, and your communications partner accepts the connection:

- Your program is in Send state for the conversation.
- Your communications partner's program is in Receive state.

You can now send data, using APPCVM SENDDATA. Your program must set up data in buffers, and the data must be in APPC logical record format. Remember that you can only send data when your program is in Send state and receive data when your program is in the Receive state.

As you send data, your communications partner is notified through one or more message pending interrupts. Your partner can then receive the data using APPCVM RECEIVE.

Ending a Conversation

When your program is finished communicating with your partner program, you should end the conversation by issuing APPCVM SEVER or IUCV SEVER.

APPCVM SEVER

You can issue an APPCVM SEVER anytime after you have established a path with your partner (that is, you must first issue a CONNECT and your partner must issue an ACCEPT). At this point your partner will receive a sever interrupt that contains information about the path and any errors that may have occurred during the sever.

You can also include log data on an APPCVM SEVER. This data can be accepted by your partner and used for debugging and error recovery.

After you issue an APPCVM SEVER, your partner can examine the sever interrupt information and:

- Issue an APPCVM SEVER to sever their side of the path.
- If log data is available, issue an APPCVM RECEIVE to obtain that data before severing using IUCV SEVER.
- Issue an IUCV SEVER to sever their side of the path.

IUCV SEVER

An IUCV SEVER can be issued at any time during a conversation. Usually your partner will receive a sever interrupt which contains information about the path and any errors that may have occurred during the sever.

After receiving an IUCV SEVER, your partner can:

- Issue an APPCVM SEVER to sever their side of the path.
- Issue an IUCV SEVER to sever their side of the path.

In a CMS environment, you can also use the CMSIUCV macro. This macro is described in the *z/VM: CMS Macros and Functions Reference*.

Using the CMS Interface to APPC/VM

You should write your APPC/VM application for a CMS environment. CMS support of APPC/VM makes it easier for multiple APPC/VM programs to operate within the same virtual machine. The CMS Shared File System, session services, private resources, CPI Communications, Coordinated Resource Recovery (CRR), and other system functions and products also require the CMS support of APPC/VM.

In CMS, you can invoke APPC/VM functions through the CMS macros named HNDIUCV and CMSIUCV. These macros are necessary to tell CMS when APPC/VM paths have been created or destroyed.

HNDIUCV and CMSIUCV macro functions have a NAME keyword and value that corresponds to one or more APPC/VM paths. This ensures that no program severs a path that another program has established. If the program requests a SEVER or an ACCEPT for a specific path and the NAME specified does not correspond with the owner of that path, the SEVER or ACCEPT is not permitted.

Note: The program name *!CMS* is reserved. CMS uses this program name so it can use APPC/VM paths.

The HNDIUCV macro functions let you:

- Initialize a program's APPC/VM environment in CMS (SET function)
- Terminate a program's APPC/VM environment in CMS (CLR function)
- Override address and information values (REP function)
- Place connection pending interrupts for private resources on a queue (HLD function)
- Release connection pending interrupts for private resources from the queue (RES function).

The HNDIUCV SET function identifies a program name to CMS. You must issue this function before issuing any CMSIUCV macro functions.

The CMSIUCV macro functions let you:

- Request to begin communications with another program in the same or different virtual machine (CONNECT function)
- Agree to begin communications with another program (ACCEPT function)
- Terminate communications with another program (SEVER function).

An APPC/VM conversation established using CMSIUCV CONNECT or CMSIUCV ACCEPT is automatically severed by CMS (using SEVER TYPE=ABEND) as part of end-of-work-unit processing. Work unit processing completes at end-of-command, end-of-subset, when DMSPURWU (purge work unit) or DMSRETWU (return work unit) routines are issued, or during an abend. If you want your conversation to remain allocated after work unit processing, you must specify this when you accept or connect to your partner.

Before issuing any CMSIUCV macro functions, your program must set up the proper parameter list with the MF=L format of the appropriate IUCV or APPCVM macro. (For instance, before issuing a CMSIUCV CONNECT, you must issue an APPCVM CONNECT with a proper parameter list and MF=L.)

You can see how APPC/VM programs use HNDIUCV and CMSIUCV functions in the following scenarios:

- "Scenario 1: Request for Global Resource" on page 295
- "Scenario 2: Request for Private Resource" on page 299.

Also, see the *z/VM: CMS Macros and Functions Reference* for detailed descriptions for each of these functions.

Note: This book does not describe how to write APPC/VM programs to run in a Group Control System (GCS) environment of z/VM. If you are writing communications applications for a GCS environment, refer to the *z/VM: Group Control System*, which contains complete descriptions of the IUCVCOM and IUCVINI macros.

CMS support of APPC/VM interrupts provided by HNDIUCV and CMSIUCV processing manage the communication linkage on conversation paths. HNDIUCV and CMSIUCV processing controls the setting of the IUCV mask (bit 30) in control register 0 for applications using this support.

Errors and Interrupts for APPC/VM in CMS

You can specify path-specific exits on the HNDIUCV and CMSIUCV macro functions. These exits are addresses of routines that receive control when an APPC/VM interrupt occurs.

Please note the following points about error conditions for APPC/VM CMS macros:

- APPC/VM generates exceptions for certain error conditions. If APPC/VM generates an operation, specification, or addressing exception while an HNDIUCV or CMSIUCV macro is executing, control does not directly return to the next sequential instruction; instead, a program check is generated.
- The CMS external interrupt handler recognizes two error conditions while an HNDIUCV or CMSIUCV macro is executing:
  - An APPC/VM connect pending external interrupt occurs and the IPRESID field does not match any active program name in the virtual machine. (An active program name is one that has been identified to CMS by HNDIUCV SET.)
  - Any other type of APPC/VM external interrupt occurs and the path that it occurs on is not owned by any active programs in the virtual machine.

  In either condition, CMS issues an IUCV SEVER for the path in error.
- If a CMS abend occurs, the following happens within the virtual machine:
  - All program names previously defined by HNDIUCV SET are made inactive.
  - External interrupt buffers created by HNDIUCV SET are destroyed.
  - Exits set up by the CMSIUCV or HNDIUCV macros are canceled.
- A program must be ready to handle any incoming external interrupts immediately after an HNDIUCV or CMSIUCV macro has finished processing. A program can even be interrupted before the sequential instruction following the macro in the program is executed.

Guidelines for Using the CMS Interface to APPC/VM

This section lists all the APPC/VM functions that can be used in an APPC/VM environment, and discusses guidelines for using these functions.

The first list shows the APPCVM and IUCV macro functions that should be directly issued from an APPC/VM program in CMS. Programs must be careful to specify a path ID when invoking each of these APPC/VM functions (except QUERY). If an APPC/VM function is invoked without specifying a particular path ID, that function will apply for the entire virtual machine. If more than one program is active in the virtual machine, the APPC/VM function could get invoked for the wrong program.

- QUERY (HNDIUCV SET also uses QUERY to determine the maximum number of connections for a virtual machine.)
- RECEIVE
- SENDCNF
- SENDCNFD
- SENDDATA
- SENDERR
- SENDREQ.

The following list describes those APPC/VM and IUCV macro functions that you should use to just set up parameter lists with MF=L.

- ACCEPT

  This should not be directly issued by a program in CMS, except to set up an IUCV ACCEPT parameter list. A CMS program should invoke the accept using CMSIUCV ACCEPT.
- CONNECT

  This should not be directly issued by a program in CMS, except to set up an APPCVM CONNECT parameter list. A CMS program should invoke the connect using CMSIUCV CONNECT.
- SEVER

  This should not be directly issued by a program in CMS, except to set up an APPCVM SEVER or IUCV SEVER parameter list. A CMS program should invoke the sever using CMSIUCV SEVER.

Some functions on the APPCVM or IUCV macro affect the APPC/VM environment of the entire virtual machine. Because CMS cannot intercept any APPCVM or IUCV macro functions directly issued by a program, any APPC/VM application program in CMS has certain limitations on its use of these functions—it must not issue any APPC/VM function that interferes with the operation of other APPC/VM applications running in the same virtual machine.

You should not use the following functions of the IUCV macro when writing CMS programs, because they could affect other APPC/VM programs in the same virtual machine:

- DCLBFR
- DESCRIBE
- RTRVBFR
- SETMASK
- SETCMASK
- TESTCMPL
- TESTMSG.

Note: CP macro libraries HCPGPI and HCPPSI are located on the system disk. The HCPGPI MACLIB contains the APPCVM macro and the IUCV macro. Refer to "Using Macro Libraries" on page 21 for more information.

Refer to "How APPC/VM Relates to General IUCV" on page 303 for detailed information about using these IUCV macro functions in an APPC/VM environment. For details about using these and any other IUCV functions in an IUCV environment, see the *z/VM: CP Programming Services*.

Managing a Resource

For a virtual machine to manage a local or global resource, it must first be authorized to connect to the Identify System Service, *IDENT. Your system administrator is the person who can authorize your virtual machine to manage a particular resource. To do this, the administrator must specify a special IUCV *IDENT statement in your virtual machine's directory entry. The *z/VM: CP Planning and Administration* describes how a system administrator authorizes virtual machines to manage resources. For details on global and local resources, see the *z/VM: CMS Macros and Functions Reference*.

Your virtual machine must connect to *IDENT before using it to manage a resource. This connect should be done using an IUCV CONNECT.

If your virtual machine becomes a local or global resource manager by establishing a connection to *IDENT, APPC/VM lets other virtual machines connect to you. The connecting virtual machines must specify, on their connection request, the resource ID you identified through *IDENT.

*IDENT maintains a local system resource table. *IDENT adds an entry to this table each time it accepts a virtual machine connection and deletes the entry when it severs the associated connection. A virtual machine manages a resource only while connected to *IDENT. For details on how *IDENT works, see the *z/VM: CP Programming Services*.

Revoking a Resource

To manage a resource, a program must identify the resource name by connecting to *IDENT. The virtual machine for the program must have proper directory authorization to connect to *IDENT. A program can also ***revoke***—stop management of—a resource name.

A program can revoke a resource it manages by issuing an IUCV SEVER to sever its path to *IDENT. *IDENT then deletes the resource from the system resource table and severs its half of the path. Your program then gets an IUCV sever interrupt from *IDENT. The SEVER does not affect existing APPC/VM paths to your virtual machine.

If another virtual machine connects to *IDENT to manage the resource that you revoked, requests to connect to the resource go to that virtual machine.

Note: If a virtual machine tries to connect to a resource that you manage before your revoke completes, the path may be established.

A program can revoke a resource that *another* program manages by issuing an IUCV CONNECT to *IDENT with the appropriate user data. The issuing program's virtual machine must have proper directory authorization to connect to *IDENT and to do this kind of revoke. Your system administrator can authorize your program's virtual machine to revoke a particular resource.

You might have a case where two disjoint TSAF collections merge, and the same resource name is identified (through connections to *IDENT) by virtual machine programs on both collections. If this happens, the TSAF virtual machine issues a revoke to one of the competing virtual machines, and *IDENT severs its path to the resource manager program within that virtual machine.

Scenario 1: Request for Global Resource

"Sequence of Instructions Requester/Server Communication" on page 296 shows the sequence of macro instructions issued when an APPC/VM assembler program in CMS communicates with a global resource manager program using APPC/VM. In this scenario, the global resource is on a different system, but within the same TSAF or CS collection.

The functions performed by these instructions include:

- Starting APPC/VM communications

- Connecting to another virtual machine
- Sending and receiving messages
- Replying to and waiting for messages
- Severing the connection to another virtual machine
- Ending APPC/VM communications.

This scenario is to give you a better idea of how APPC/VM programs work together. Note that the macro functions can be more detailed than what is shown in this scenario.

Sequence of Instructions Requester/Server Communication

```
User Program                      Global Resource Manager
                                  Program
(Requester Virtual Machine)       (Server Virtual Machine)
-----------------------------     --------------------------
1 HNDIUCV SET,NAME=USRNAME,       1 HNDIUCV SET,NAME=RESNAME,
    EXIT=X1                          EXIT=X2
                                  2 IUCV CONNECT PRMLIST=PLADDR,
                                      USERID=*IDENT,
                                      USERDTA=stuff,MF=L
                                  3 CMSIUCV CONNECT NAME=RESNAME,
                                      PRMLIST=PLADDR,EXIT=X2,
                                      ERROR=ERR2
                                  4 gets connect complete
                                      interrupt

5 APPCVM CONNECT,PRMLIST=PLADDR,
    MF=L,RESID=RESNAME,WAIT=NO,
    BUFLEN=0                       program waits for interrupt
6 CMSIUCV CONNECT,NAME=USRNAME,                      .
    PRMLIST=PLADDR,EXIT=X1,                          .
    COMDIR=NO,ERROR=ERR1

            .                     7 connect-pending external
                                      interrupt
            .                     8 exit X2 receives control
                                      and posts ECB
program busy waits for interrupt  9 IUCV ACCEPT,PRMLIST=PLADDR,
                                     PATHID=path2,MF=L
        .                         10 CMSIUCV ACCEPT,NAME=RESNAME,
        .                             PRMLIST=PLADDR,EXIT=X2
                                                     .
11 gets connect complete interrupt                   .
12 exit X1 gets control, posts ECB program waits for interrupt
13 APPCVM SENDDATA PRMLIST=PLADDR,                     .
     PATHID=path1,BUFFER=dataddr,                      .
     RECEIVE=NO                                        .

                                  14 gets message pending
                                       interrupt
                                  15 exit X2 receives control
                                       and posts ECB
                                  16 APPCVM RECEIVE
                                       PRMLIST=PLADDR,
                                       PATHID=PTH2
                                       BUFFER=bufaddr,
                                       BUFLEN=length


17 X1 gets control for FCI
        .                                              .
               18  sending and receiving continue
        .                                              .
        .                                              .
19 APPCVM SEVER,PRMLIST=PLADDR,
     TYPE=NORMAL,PATH=path1,MF=L
20 CMSIUCV SEVER,NAME=USRNAME,
     PRMLIST=PLADDR, CODE=ONE
21 X1 gets control for FCI
                                  22 sever interrupt
                                  23 exit X2 receives control
                                  24 APPCVM SEVER,PRMLIST=PLADDR,
                                       PATHID=path2,MF=L
                                  25 CMSIUCV SEVER,NAME=RESNAME,
                                       PRMLIST=PLADDR
                                  26 X2 gets control for FCI

27 HNDIUCV CLR,NAME=USRNAME       27 HNDIUCV CLR,NAME=RESNAME
        .                         .
        .                         .
        .                         .
28  X1  address of exit routine   X2  address of exit routine
29  ERR1  address of error        ERR2 address of error
          routine                      routine
30  USRNAME  DC   CL8'RED'        RESNAME  DC   CL8'BLUE'
31  RESNAME  DC   CL8'BLUE'
32  PLADDR   DS XL40              PLADDR   DS XL40
```

The following list explains the sequence of instructions shown in "Sequence of Instructions Requester/ Server Communication" on page 296.

1. Each program must first independently issue HNDIUCV SET to declare itself to CMS. (This function creates a buffer for CP to store external interrupt data.) Each program also specifies an exit address that gets control in case of an interrupt.
2. The resource manager program must establish itself as a resource manager by connecting to *IDENT. This allows APPC/VM connections from user programs. In this step, the program just sets up the IUCV CONNECT parameter list using MF=L.
3. The program issues a CMSIUCV CONNECT to actually invoke the connection to *IDENT.
4. The resource manager program receives a connection complete interrupt as a result of the connect request to *IDENT.
5. The user program wants to connect to resource BLUE (referenced by label RESNAME). It must identify this resource on the RESID parameter. In this step, the user program just sets up an APPCVM CONNECT parameter list using MF=L.

   Note:

   a. This RESID value must match the name that the resource manager program specified on its HNDIUCV SET. In this example, the RESID is the value *BLUE*.
   b. Specifying BUFLEN=0 means that we are not supplying a connection parameter list extension.
6. The user program issues a CMSIUCV CONNECT to actually invoke the connection to the resource manager program and create a path (from the user program's side). The exit address, X1, is associated with the path ID. Specifying COMDIR=NO means that no communications directory resolution has to be done; *BLUE* is the only value necessary to establish the connection.
7. The resource manager program receives a connection pending interrupt as a result of the user program's connect request.
8. The interrupt (exit) routine at address X2 receives control as a result of the external interrupt. (X2 receives control because it was specified on the EXIT parameter of the HNDIUCV SET macro.) The interrupt routine posts the ECB, then returns control to the "mainline" resource manager program.
9. The resource manager program wants to do an ACCEPT to complete the connection. In this step, it issues an IUCV ACCEPT with MF=L to format an IUCV ACCEPT parameter list.
10. The resource manager program issues a CMSIUCV ACCEPT to actually invoke the ACCEPT. This completes the APPC/VM communications link with the user program and creates a path (from the resource manager program's side). The EXIT parameter associates the address, X2, with the path ID.
11. The user program is presented with a connection-complete external interrupt as a result of the resource manager's CMSIUCV ACCEPT.
12. The exit routine at address X1 receives control as a result of the external interrupt. (X1 receives control because it is specified on the EXIT parameter of the CMSIUCV CONNECT.)
13. The user program sends data to the resource manager by issuing an APPCVM SENDDATA. The data being sent would be located at the address specified on BUFFER.
14. The resource manager program gets a message pending interrupt because of the user program's SENDDATA.
15. The interrupt (exit) routine at X2 receives control, posts the ECB, then returns control to the mainline resource manager program.

16. The resource manager receives the amount of data pending by issuing one or more APPCVM RECEIVE functions.
17. The interrupt (exit) routine at X1 receives control for a function complete external interrupt, notifying the user program of the completion of the APPCVM SENDDATA issued in step 13.
18. Data is sent and received back and forth between the two programs using APPCVM SENDDATA and APPCVM RECEIVE functions.

    Note: When a program specifies APPCVM SENDDATA RECEIVE=NO, it is in Send state and its partner is in Receive state. However, a program issuing an APPCVM SENDDATA can switch conversation states by specifying RECEIVE=YES on the SENDDATA, or by issuing the APPCVM RECEIVE function.
19. The user program completes its communications with the resource manager and wants to terminate its APPC/VM communications path. It sets up the APPCVM SEVER parameter list.
20. The user program then issues a CMSIUCV SEVER to actually invoke the SEVER function. This also clears the exit address associated with the communications path (specified on CMSIUCV CONNECT).
21. The interrupt (exit) routine at X1 receives control for a function complete external interrupt, notifying the user program of the completion of the APPCVM SEVER issued in step 19.
22. The resource manager receives a sever external interrupt as a result of the CMSIUCV SEVER issued by the source program.
23. The exit routine at address X2 gets control as a result of the external sever interrupt.
24. The resource manager program wants to terminate the communications link. In this step, it sets up an APPCVM SEVER parameter list.
25. The resource manager issues CMSIUCV SEVER to actually invoke the sever. This clears the exit address associated with the communications path (specified on the CMSIUCV ACCEPT).
26. The interrupt (exit) routine at X2 receives control for a function complete external interrupt, notifying the resource manager program of the completion of the APPCVM SEVER issued in step 24.
27. After all communications are complete and the communications path has been severed on each side, the two programs independently issue HNDIUCV CLR. HNDIUCV CLR terminates APPC/VM communications for the program name, and clears the general exit address (specified on HNDIUCV SET).
28. This is the label of the exit (interrupt handler) routine. This routine receives control when an interrupt occurs on the associated path. The routine checks for various types of interrupts, post an ECB, then return control to the main program.
29. This is the label of an error routine. If an error is encountered while processing the associated function, then this routine receives control.
30. This is the name that the program must first identify to CMS on HNDIUCV SET and then on any subsequent HNDIUCV or CMSIUCV macros. The user program has a name of RED, and the resource manager program has a name of BLUE.
31. The user program must specify the resource manager program name, BLUE (resolved through COMDIR), as the resource ID (RESID) on the APPCVM CONNECT.
32. This declares 40-byte parameter list areas.

Use the z/VM HELP Facility for complete details on all APPC/VM functions, or use the following publications:

- For details on the APPCVM and IUCV macro functions, see the *z/VM: CP Programming Services*.
- For complete details on all CMSIUCV and HNDIUCV macro functions, see the *z/VM: CMS Macros and Functions Reference*.

Scenario 2: Request for Private Resource

"Sequence of Steps in Private Resource Request Processing" on page 299 shows the sequence of steps that occur when an APPC/VM assembler user program in CMS requests a private resource from an APPC/VM assembler private resource manager program.

In this scenario, the private resource is on a different system, but within the same TSAF collection. Assume the requester virtual machine has a user ID of "REQ1" and the server virtual machine has a user ID of "SERV1".

The functions performed in this scenario include:

- Setting up the two communicating virtual machines
- Starting APPC/VM communications
- Connecting to another virtual machine
- Sending and receiving messages
- Replying to and waiting for messages
- Severing the connection to another virtual machine
- Ending APPC/VM communications.

This scenario is to give you a better idea of how APPC/VM programs work together. Note that the macro functions can be more detailed than what is shown in this scenario.

Sequence of Steps in Private Resource Request Processing

```
   User Program                     Private Resource Manager Program          Program
 (Requester Virtual Machine)         (Server Virtual Machine)
 -----------------------------      -------------------------------           -------
Enable communications directory file  Set up $SERVER$ NAMES file

1 HNDIUCV SET,NAME=USRNAME,EXIT=X1
2 APPCVM CONNECT,PRMLIST=PLADDR,
     RESID=RESNAME,WAIT=NO,
     BUFLEN=0,MF=L
3 CMSIUCV CONNECT,NAME=USRNAME,
     PRMLIST=PLADDR,EXIT=X1,
     COMDIR=YES,ERROR=ERR1

                                      4 CMS invokes program
                                      5 HNDIUCV SET,NAME=RESNAME,
                                        EXIT=X2
            .                         6 exit X2 receives control
                                        and posts ECB
   program waits for interrupt        7 IUCV ACCEPT,
                                        PRMLIST=PLADDR,
                                        PATHID=path2,MF=L
        .                             8 CMSIUCV ACCEPT,
                                        NAME=RESNAME,
        .                               PRMLIST=PLADDR,EXIT=X2
                                      9 HNDIUCV HLD,NAME=RESNAME
                                                     .

10 gets connect complete interrupt                   .
11 exit X1 gets control and posts ECB  program waits for interrupt
12 APPCVM SENDDATA PRMLIST=PLADDR,                   .
     PATHID=path1,BUFFER=dataddr,                    .
     RECEIVE=NO                                      .
                                      13 gets message pending
                                         interrupt
                                      14 exit X2 receives control
                                         and posts ECB
                                      15 APPCVM RECEIVE
                                         PRMLIST=PLADDR,
                                         PATHID=path2
                                         BUFFER=bufaddr,
                                         BUFLEN=length

16 X1 gets control for FCI
        .                                            .
        .                                            .
```

```
                   17  sending and receiving continue
        .                                              .
        .                                              .
18 APPCVM SEVER,PRMLIST=PLADDR,
     TYPE=NORMAL,PATH=path1,MF=L
19 CMSIUCV SEVER,NAME=USRNAME,
     PRMLIST=PLADDR, CODE=ONE
20 X1 gets control for FCI

                                      21 sever interrupt
                                      22 exit X2 receives control
                                      23 APPCVM SEVER,
                                         PRMLIST=PLADDR,
                                         PATHID=path2,MF=L
                                      24 CMSIUCV SEVER,
                                         NAME=RESNAME,
                                         PRMLIST=PLADDR
                                      25 X2 gets control for FCI
                                      26 HNDIUCV RES,
                                         NAME=RESNAME,
                                         PRMLIST=PLADDR

27 HNDIUCV CLR,NAME=USERNAME          27 HNDIUCV CLR,NAME=RESNAME
        .                                 .
        .                                 .
        .                                 .
28  X1    address of exit routine         X2 address of exit
                                             routine
29  ERR1  address of error routine        ERR2 address of error
                                               routine
30  USRNAME  DC   CL8'RED'                RESNAME  DC   CL8'BLUE'
31  RESNAME  DC   CL8'TARGET'
32  PLADDR   DS XL40                      PLADDR   DS XL40
```

Use the HELP Facility for complete details on all APPC/VM functions, or use the following publications:

- For details on the APPCVM and IUCV macro functions, see the *z/VM: CP Programming Services*.
- For complete details on all CMSIUCV and HNDIUCV macro functions, see the *z/VM: CMS Macros and Functions Reference*.

Virtual Machine Preparations

- The requester virtual machine should have a communications directory entry set up so it can connect to the private resource manager. The communications directory file might have an entry that looks like this:

  ```
  :nick.TARGET       :tpn.BLUE
                     :luname.*USERID SERV1
                     :security.NONE
  ```

  Note that the transaction program name entry must match the name that the resource manager program specified on its HNDIUCV SET. Also note that no access security information is needed for this particular connection.
- The private server virtual machine program must set up a $SERVER$ NAMES file so it can check to see if a requesting program is authorized. The name of the resource is *BLUE* and the user ID of the requesting virtual machine is "REQ1". However, because our private server will not check security, the $SERVER$ NAMES file has an entry that looks like this:

  ```
  :nick.BLUE     :list.*
  ```

Program Functions

1. The user program must issue HNDIUCV to declare itself to CMS. (This function creates a buffer for CP to store external interrupt data.) This function also specifies an exit routine address that gets control in case of an interrupt.
2. The user program wants to connect to resource that has a TPN of *BLUE*. It must identify this resource on the RESID parameter; this RESID value is a symbolic destination name, used as an index to the

CMS communications directory file. In this step, the user program just sets up an APPCVM CONNECT parameter list using MF=L.

The user program does not have to specify a connection parameter list extension, so it specifies BUFLEN=0. The extension will be automatically created and filled in from information in the communications directory. (This is because COMDIR=YES is specified in step 3.)

3. The user program issues a CMSIUCV CONNECT to actually invoke the connection to the resource manager program and create a path (from the user program's side). The exit address, X1, is associated with the path ID.
4. One of the following scenarios must occur for a successful private resource connection:
   a. If the private resource virtual machine is already logged on, CMS must be IPLed and the SET SERVER ON command must have been issued.
   b. If the private resource virtual machine is not already logged on,
      i) CP must be able to autolog it.
      ii) The directory for this private resource manager virtual machine must contain an IPL CMS statement.
      iii) The virtual machine's PROFILE EXEC, which is automatically invoked during an IPL CMS, must contain the SET SERVER ON command.

   Also, in each case, FULLSCREEN must be set OFF.

   CP presents an APPC/VM connection pending external interrupt to CMS for the private resource named BLUE. CMS validates the user ID presented in the connection pending interrupt by checking the $SERVER$ NAMES file for this virtual machine. Because * was specified for BLUE, the user ID is automatically authorized. CMS then queues the connection pending interrupt for the private server.

   When CMS is in the `Ready;` state for the private server, CMS invokes the specified private resource program. The name of the program being invoked is either the resource ID (default), or the value of the :module. tag in a matching $SERVER$ NAMES file entry if one exists.
5. The private resource manager program issues an HNDIUCV SET to identify itself to CMS as an APPC/VM program. You must also specify a user exit address on the HNDIUCV SET macro. Control is passed to this user exit address when an APPC/VM or IUCV connection pending interrupt is received for the resource ID.

   If there are any queued connection pending interrupts for this private resource when the HNDIUCV SET macro is issued, control is passed to the user exit address before the instruction following the HNDIUCV macro is executed.
6. CMS takes the interrupt buffer off the queue (with a resource ID *BLUE*, which matches the NAME parameter on HNDIUCV) and passes control to the user interrupt processing exit routine at label "X2".

   (At this point, the resource manager could issue an APPCVM RECEIVE to receive the allocate data; CP purges this data if the program does not receive it before accepting the connection.)
7. The resource manager program wants to do an ACCEPT to complete the connection. In this step, it issues an IUCV ACCEPT with MF=L to format an IUCV ACCEPT parameter list.
8. The resource manager program issues a CMSIUCV ACCEPT to actually invoke the ACCEPT. This completes the APPC/VM communications link with the user program and creates a path (from the resource manager program's side). The EXIT parameter associates the address, X2, with the path ID.
9. The resource manager issues an HNDIUCV HLD for the resource (program) name *BLUE*. This causes CMS to queue any subsequent connection pending interrupts for *BLUE*.
10. The user program is presented with a connection-complete external interrupt as a result of the resource manager's CMSIUCV ACCEPT.
11. The exit routine at address X1 receives control as a result of the external interrupt. (X1 receives control because it is specified on the EXIT parameter of the CMSIUCV CONNECT.)

12. The user program sends data to the resource manager by issuing an APPCVM SENDDATA. The data being sent would be located at the address specified on BUFFER.
13. The resource manager program gets a message pending interrupt because of the user program's SENDDATA.
14. The interrupt (exit) routine at X2 receives control, posts the ECB, then returns control to the mainline resource manager program.
15. The resource manager receives the amount of data pending by issuing one or more APPCVM RECEIVE functions.
16. The interrupt (exit) routine at X1 receives control for a function complete external interrupt, notifying the user program of the completion of the APPCVM SENDDATA issued in step 12.
17. Data is sent and received back and forth between the two programs using APPCVM SENDDATA and APPCVM RECEIVE functions.

    Note: When a program specifies APPCVM SENDDATA RECEIVE=NO, it is in Send state and its partner is in Receive state. However, a program issuing an APPCVM SENDDATA can switch conversation states by specifying RECEIVE=YES on the SENDDATA, or by issuing the APPCVM RECEIVE function.
18. The user program completes its communications with the resource manager and wants to terminate its APPC/VM communications path. It sets up the APPCVM SEVER parameter list.
19. The user program then issues a CMSIUCV SEVER to actually invoke the SEVER function. This also clears the exit address associated with the communications path (specified on CMSIUCV CONNECT).
20. The interrupt (exit) routine at X1 receives control for a function complete external interrupt, notifying the user program of the completion of the APPCVM SEVER issued in step 18.
21. The resource manager receives a sever external interrupt as a result of the CMSIUCV SEVER issued by the source program.
22. The exit routine at address X2 gets control as a result of the external sever interrupt.
23. The resource manager program wants to terminate the communications link. In this step, it sets up an APPCVM SEVER parameter list.
24. The resource manager issues CMSIUCV SEVER to actually invoke the sever. This clears the exit address associated with the communications path (specified on the CMSIUCV ACCEPT).
25. The interrupt (exit) routine at X2 receives control for a function complete external interrupt, notifying the resource manager program of the completion of the APPCVM SEVER issued in step 23.
26. The resource manager issues HNDIUCV RES to remove the hold status for resource name *BLUE*. This means that the resource manager program is ready to handle the first queued connection pending interrupt.
27. After all communications are complete and the communications path have been severed on each side, the two programs independently issue HNDIUCV CLR. HNDIUCV CLR terminates APPC/VM communications for the program name, and clears the general exit address (specified on HNDIUCV SET).
28. This is the label of the exit (interrupt handler) routine. This routine receives control when an interrupt occurs on the associated path. The routine checks for various types of interrupts, post an ECB, then return control to the main program.
29. This is the label of an error routine. If an error is encountered while processing the associated function, then this routine receives control.
30. This is the name that the program must first identify to CMS on HNDIUCV SET and then on any subsequent HNDIUCV or CMSIUCV macros. The user program has a name of RED, and the resource manager program has a name of BLUE.
31. The user program must specify the resource manager program name, BLUE, as the resource ID (RESID) on the APPCVM CONNECT.
32. This declares 40-byte parameter list areas.

How APPC/VM Relates to General IUCV

The Inter-User Communications Vehicle (IUCV) provides a way for program-to-program communications within one VM system. A program using IUCV can communicate with itself, with a CP system service, or with another program on the same system. IUCV is not part of the APPC (SNA LU 6.2) architecture. For details on IUCV, see the *z/VM: CP Programming Services*.

APPC/VM, which is VM's implementation of APPC (SNA LU 6.2) protocol, includes some IUCV support. However, because IUCV is not part of the APPC architecture, it is important to know the differences between APPC/VM and IUCV.

To start, APPC/VM depends on a half-duplex protocol, while IUCV communication uses a full-duplex protocol. In support of half-duplex protocol, APPC/VM defines and enforces states on each path.

Also, in APPC/VM the high-order bit of the IPTYPE field is set to designate APPC/VM from IUCV interrupts. (The IPTYPE field is part of the interrupt information.)

APPC/VM and IUCV each provide a set of communication functions. The following list shows the differences and similarities between APPC/VM functions and IUCV functions.

- **APPC/VM and IUCV functions that work differently**:
  - CONNECT
  - RECEIVE
  - SEND (for APPC/VM, SENDDATA).

  Specific differences on APPCVM macro and IUCV macro functions are described in the *z/VM: CP Programming Services*.
- **IUCV functions not supported on APPC/VM paths**:
  - PURGE
  - QUIESCE
  - RESUME
  - REJECT (APPCVM SENDERR is similar, however.)
  - REPLY (APPCVM SENDDATA can be used instead, however.)
- **APPC/VM functions not supported on IUCV paths**:
  - SENDCNF and SENDCNFD

    IUCV has no equivalent functions.
  - SENDERR

    IUCV REJECT is similar, however.
  - SENDREQ

    However, an IUCV priority 1WAY parameter data SEND is similar when that SEND is used as a signal and does not contain any data.
- **Shared APPC/VM and IUCV functions**

  The following IUCV macro functions can be used on APPC/VM and IUCV paths, and can be used safely in a CMS environment:
  - ACCEPT
  - QUERY
  - SEVER.

  Other IUCV macro functions can be used on APPC/VM and IUCV paths, but should not be used in a CMS environment because they could affect other programs in the same virtual machine.

  These functions are:

- **DCLBFR**

  DCLBFR declares an interrupt buffer. (Both APPC/VM and IUCV interrupts are presented in the same buffers.)

  DCLBFR should not be directly issued by a program in CMS; HNDIUCV uses DCLBFR to initialize the virtual machine's APPC/VM environment.

- **DESCRIBE**

  DESCRIBE gives the following information:

  - The next message pending on non-APPC paths
  - The next message pending on an APPC path that is in Receive state
  - A SENDREQ on an APPC path.

  DESCRIBE should not be used in CMS because this function clears the pending-message external interrupt for the described message. This interrupt may not belong to the issuer of the DESCRIBE function; thus, other programs running in the same virtual machine can be affected because the message is lost and never reflected to the true target.

- **RTRVBFR**

  RTRVBFR releases an interrupt buffer. (Both APPC/VM and IUCV interrupts are presented in the same buffers.)

  RTRVBFR should not be directly issued by a program in CMS; HNDIUCV and CMS abend processing use RTRVBFR to terminate a virtual machine's APPC/VM environment.

- **SETMASK** and **SETCMASK**

  SETMASK and SETCMASK disable and enable APPC and non-APPC interrupts.

  These functions should not be used by a program in CMS because they disable certain APPC/VM external interrupts for the entire virtual machine. Thus, other programs running in the same virtual machine may be affected.

- **TESTCMPL**

  TESTCMPL determines the next APPC or non-APPC function that has completed.

  TESTCMPL can be directly issued by a program in CMS; however, the issuer must be careful that a message ID or path ID is specified in the IUCV parameter list. If it is not, APPC/VM completes the first message on the REPLY queue for the entire virtual machine, and that message may not belong to the application that issued the TESTCMPL.

- **TESTMSG**

  TESTMSG waits for the following:

  - A message pending or message complete interrupt on non-APPC paths
  - A message pending interrupt on an APPC path that is in Receive state
  - A request-to-send interrupt on an APPC path
  - A function complete interrupt on an APPC path.

  TESTMSG should not be used by a program in CMS because it places the entire virtual machine in a wait state if no incoming messages or replies are pending. Thus, other programs running in the same virtual machine may be affected.

CMS Support of APPC/VM

CMS provides an interface to APPC/VM through two macros called CMSIUCV and HNDIUCV. Usually CMSIUCV macros are used for creating and destroying paths and HNDIUCV macros are used for identifying programs to CMS. CMSIUCV and HNDIUCV are similar to their IUCV counterparts but they tell CMS about APPCVM paths that have been created or destroyed.

Two parameters that CMS provides with the CMSIUCV and HNDIUCV macros include **ERROR** and **EXIT**. All of the CMSIUCV and HNDIUCV macros let you specify the ERROR parameter which gives the address of a routine that receives control when an error occurs. Most of the CMSIUCV and HNDIUCV macros let you to specify EXIT which gives the address of an exit routine to receive control whenever an APPC/VM external interrupt occurs on an APPV/VM path.

The following steps show you the sequence of macros you and your partner can use to set up an APPC/VM conversation.

1. HNDIUCV SET - Identifies a program to CMS.
2. APPCVM CONNECT - Sets up an APPCVM connection parameter list using MF=L.
3. CMSIUCV CONNECT - Requests that CMS perform the connect. The COMDIR parameter on this macro lets you indicate that you want communication directory resolution.
4. IUCV ACCEPT - Sets up an ACCEPT parameter list using MF=L.
5. CMSIUCV ACCEPT - Requests that CMS perform the accept.
6. APPCVM SEVER - Sets up the APPCVM SEVER parameter list using MF=L.
7. CMSIUCV SEVER - Requests that CMS perform the sever.
8. HNDIUCV CLR - Removes an APPC/VM program from the list of active APPC/VM programs in CMS.

Summary of APPC/VM Assembler Macro Functions

The following table summarizes APPC/VM assembler macro functions. Details on APPCVM and IUCV macro functions are described in the *z/VM: CP Programming Services*. Details on CMSIUCV and HNDIUCV macro functions are contained in *z/VM: CMS Macros and Functions Reference*.

Table 27. Summary of APPC/VM Assembler Macro Functions

Function	Macro	Description
ACCEPT	IUCV	Accepts the connection from a requesting program to complete a path.
ACCEPT	CMSIUCV	Accepts the connection from a requesting program to complete a path, and lets CMS know about it.
CLEAR	HNDIUCV	Removes an APPC/VM program name from the list of APPC/VM programs that are active in CMS.
CONNECT	APPCVM	Establishes and reserves a path to communicate with another program.
CONNECT	CMSIUCV	Establishes and reserves a path to communicate with another program, and lets CMS know about the connection.
CONNECT	IUCV	Establishes and reserves a path for resource manager programs to communicate with *IDENT.
HOLD	HNDIUCV	Temporarily places private resource connection requests on a CMS queue.
QUERY	IUCV	Gets information about the external interrupt buffer and finds out how many paths can be established.
QCMSWID	CMSIUCV	Gets the current CMS work unit identifier.
RECEIVE	APPCVM	Receives data and information sent to your program.
REPLACE	HNDIUCV	Replaces the exit address and UWORD for APPC/VM programs that have been declared to CMS.

Table 27. Summary of APPC/VM Assembler Macro Functions (continued)

Function	**Macro**	**Description**
RESOLVE	CMSIUCV	Gets values from a CMS communications directory file for examination.
RESUME	HNDIUCV	Releases previously-held private resource connection requests from a CMS queue.
SENDCNF	APPCVM	Sends a confirmation request to your communications partner.
SENDCNFD	APPCVM	Sends a response to a confirmation request.
SENDDATA	APPCVM	Sends data to your communications partner.
SENDERR	APPCVM	Sends notice to your communications partner that your program has detected an error.
SENDREQ	APPCVM	Requests permission to send data.
SET	HNDIUCV	Declares an APPC/VM program name to CMS.
SEVER	APPCVM	Ends communications with another program.
SEVER	CMSIUCV	Ends communications with another program, and lets CMS know about it.
SEVER	IUCV	Ends communications with another program when APPCVM SEVER is not appropriate.

Chapter 21. Using Advanced APPC/VM Functions

In addition to the basic functions, APPC/VM also provides more advanced functions that you can use in your programs. These advanced functions allow you to confirm communications with your partner program, send error indications to your partner, ask your partner for permission to start sending data, and synchronize your functions.

Requesting Confirmation

When you are sending data to another virtual machine, you may want to confirm that you should continue to send. To do this, you issue the SENDCNF function and specify TYPE=NORMAL. The SENDCNF is not complete until your partner issues one of the following functions:

- SENDCNFD to indicate that the sender can continue
- SENDERR to indicate that something is wrong
- SEVER to end communications.

When you are sending data to another virtual machine, you may want to sever the connection. You can confirm this decision with your communications partner by issuing the SENDCNF function and specify TYPE=SEVER.

To use SENDCNF and SENDCNFD, the program starting the conversation must indicate that confirmation is allowed on the conversation. The program does this by specifying SYNCLVL=CONFIRM on the APPCVM CONNECT function.

Signaling an Error

When you sense that there is an error in the communications, whether you are in Send or Receive state, you can issue SENDERR. This signals your communications partner and causes a break in the normal send/receive sequence.

If you are in Receive state, and issue SENDERR:

1. The error notice goes to your communications partner.
2. Your virtual machine enters Send state when the SENDERR completes.

If you are in Send state, and issue SENDERR:

1. The error notice goes to your communications partner.
2. Your virtual machine remains in Send state.

When sending an error notice to your communications partner, you can also send log data. Log data contains information that describes the error in detail, and it can help your partner diagnose the error. For log data to get to your partner, however, your partner must have indicated that it will accept log data. This can be done using the LOGDATA=YES parameter for IUCV ACCEPT. Note that you can also send log data on an APPCVM SEVER.

Requesting to Send

At some point, when your partner is sending data, you may want to interrupt your partner so that you can send data. To do this, you issue a SENDREQ, which is presented to your partner as a request-to-send interrupt.

Your program will be able to send data if your partner decides to change states (by issuing RECEIVE or SENDDATA RECEIVE=YES) because of your SENDREQ However, note that your partner can choose to ignore your SENDREQ.

Sending and Receiving Early Information

When issuing an APPCVM CONNECT, you can send data to your communications partner that the partner can use *before it accepts your connection*.

This early data is referred to as **program initialization parameters**, or PIP data. Regular application programs can set up PIP data in storage and then specify the data's address and length in the APPCVM CONNECT parameter list extension. PIP data is sent to a partner program through a PIP variable.

Here are some possible uses for sending PIP data:

- The target program can use the information to initialize values.
- A source program could send an entire *start-up* program to a target.
- In a distributed environment, a central program can send identical loader information to all remote nodes.

Target programs know if PIP data is present on an incoming connection by looking at a field (IPPIPLEN) in the connection pending interrupt. The target program can obtain information in a PIP variable by specifying APPCVM RECEIVE PIP=YES before accepting the connection. Programs running on CMS and using CMS support for communication do not need to issue a receive to get a PIP variable. CMS receives the PIP variable and places its address in register 4.

Communications servers must set up a special area when forwarding PIP data for a requesting application. This area is called a VM communications server area. For more information on PIP data in z/VM, see the *z/VM: CP Programming Services*.

Using Synchronous Functions

When you issue a ***synchronous*** APPC/VM function (by specifying WAIT=YES on the APPCVM macro), your virtual machine goes into a WAIT state. This means you cannot issue any APPC/VM functions to any paths until the synchronous function that you issued completes. You can, however, enter the CP commands IPL, LOGOFF, SYSTEM RESET, and SYSTEM CLEAR to terminate the wait state.

When a synchronous APPC/VM function completes you do not get an interrupt to tell you about the completion; instead, the function complete data goes to the output parameter list for the function.

The following can cause your APPC/VM function to complete:

- Your partner issues a function to complete your function
- Your partner logs off or resets its virtual machine
- You log off or reset your virtual machine
- Your partner completes work unit processing for APPC/VM conversations established using CMSIUCV CONNECT or CMSIUCV ACCEPT
- You complete work unit processing for APPC/VM conversations established using CMSIUCV CONNECT or CMSIUCV ACCEPT.

Synchronous functions are not affected by interrupts of any kind. Even if your virtual machine is enabled for interrupts, you are not given control until your function completes.

Use synchronous functions carefully! If your communications partner does not respond, log off, or do a system reset, your virtual machine cannot execute any instruction until you IPL, log off, reset, or clear your virtual machine. Applications should take responsibility to avoid deadlock situations. A deadlock situation is when two virtual machines are waiting for an action or response from each other. Do not use the synchronous option for a program that must serve more than one user at the same time, or for a program that must run in a multitasking environment within a virtual machine.

Using ***asynchronous*** APPC/VM functions can help you avoid a deadlock situation. An APPC/VM function that is asynchronous (WAIT=NO is specified) may or may not complete immediately:

- When the APPC/VM function does not complete immediately, your virtual machine must wait before issuing other APPC/VM functions on the path; however, you can issue APPC/VM functions to *other* paths while waiting for the asynchronous function to complete. You get an interrupt when the function does complete.
- When an asynchronous APPC/VM function does complete immediately, you do not receive a function complete interrupt; however, you get the function complete data in the output parameter list.

Your communications partner sees no difference if you issue APPC/VM functions synchronously (WAIT=YES) or asynchronously (WAIT=NO).

Note: The asynchronous capability of APPC/VM is a VM-unique implementation, it is not based on APPC (SNA LU 6.2) architecture.

Synchronizing Updates to Multiple Resources

You can write an APPC/VM application that uses CMS's Coordinated Resource Recovery (CRR) to coordinate updates to multiple participating resources. CRR ensures that either all updates are made together, or no updates are made. This can help you to protect against network failure and other data integrity problems.

Suppose that two application programs, APPLA and APPLB, want to establish a conversation and:

- APPLA wants to make updates to RESA
- APPLB wants to make updates to RESB
- APPLA and APPLB want to make sure the updates are made as a unit.

To accomplish this, APPLA must establish a protected conversation with APPLB by:

1. Setting up the parameter list using APPCVM CONNECT
2. Using CMSIUCV CONNECT to do the connection.

Because you want updates to RESA and RESB to be performed in unison, APPLA must specify SYNCLVL=SYNCPT on the APPCVM CONNECT. A conversation established with SYNCLVL=SYNCPT is known as a ***protected conversation***. Using a protected conversation tells CMS that the conversation between APPLA and APPLB is part of a ***logical unit of work***. See the *z/VM: CMS Application Development Guide* for a discussion of CMS work units and the concept of logical units of work.

After making updates to RESA and RESB, APPLA (or APPLB) can choose to either ***commit*** the changes or ***roll back*** the changes. The commit (or rollback) marks the boundary between logical units of work.

A commit is done using the CSL routine DMSCOMM or the SAA resource recovery (also known as CPI resource recovery) routine, SRRCMIT. A rollback is done using the CSL routine DMSROLLB or the SAA resource recovery routine, SRRBACK. These routines are fully described in the .

If an APPC/VM function completes with IPWHATRC=IPPREPAR or IPREQCOM, your partner has initiated a commit. After preparing the other resources associated with that work unit, you should commit (or rollback) the work unit.

If an APPC/VM function completes with IPWHATRC=IPBACK, your partner has initiated a rollback. You should roll back the work unit associated with that conversation.

If you sever (using CMSIUCV SEVER or HNDIUCV CLR) or your partner severs (an APPC/VM function completes with IPWHATRC=IPSABEND or IPSNORM) a protected conversation, you should roll back the work unit associated with that conversation before doing any other processing for that work unit because it is likely that your work unit needs to be rolled back.

If a work unit requires a rollback, all protected resources associated with that work unit, including APPC/VM protected conversations, are forced into a state that requires a rollback. As a result, you can only issue a rollback or one of the following APPCVM functions at this time:

- CMSIUCV SEVER with either an IUCV SEVER parameter list or an APPCVM SEVER parameter list with TYPE=ABEND
- HNDIUCV CLR

- CMSIUCV SEVER with an APPCVM SEVER parameter list with TYPE=NORMAL when the conversation is in `sever` state
- APPCVM QRYSTATE
- APPCVM RECEIVE if log data is pending.

Issuing any other APPCVM function will result in a state check.

It is possible, however, that the work unit does not require a rollback. For example, if the last sync point resulted in a rollback, it is possible that the conversation was severed during that sync point and so the work unit does not require another rollback. A procedure for determining if a rollback is required follows.

Determining When a Sever Requires a Rollback

1. If your partner severs:
   a. If IPWHATRC=IPSABEND and IPCODE=X'01*xx*:X', this is an allocation error. A rollback is not required.
   b. If you issue an APPC/VM function that completes with IPWHATRC=IPBACK, then a rollback is required.
   c. If you issue an APPC/VM function that initially completes with CC=0 and subsequently completes with a function complete interrupt with IPWHATRC=IPSABEND or IPSNORM, then a rollback is required.
   d. If you issued an APPC/VM function while disabled for external interrupts and the function completes immediately (CC=2) with IPWHATRC=IPSABEND or IPSNORM, then (while still disabled):
      i) APPCVM RECEIVE any log data
      ii) Set up an APPC/VM Sever or IUCV SEVER parameter list
      iii) Use CMSIUCV SEVER to sever the path and proceed with "2" on page 310 below.
   e. If you issue an APPC/VM function while enabled for external interrupts and the function completes immediately (CC=2) with IPWHATRC=IPSABEND or IPSNORM, then see "3" on page 310 below to determine whether the work unit requires a rollback.
2. If you initiate the sever or if you sever in response to a sever from your partner (see "1.d" on page 310 above), the sever you issued may merely have freed the path ID of a conversation that was severed in the preceding sync point that resulted in a rollback. You can determine if this is the case by checking the APPCVM or IUCV SEVER output parameter list: If the IPFREPTH flag is 0 or the IPURGBKR flag is 1, then a rollback is required. Otherwise, a rollback is not required.
3. The only other way to determine if a rollback is required is to query the conversation state (APPCVM QRYSTATE) of some other conversation (if any) that is associated with the same work unit. If IPSTATE=IPBKREQ, the associated work unit requires a rollback.
4. If protected conversations are severed due to HNDIUCV CLR, you can take one of the following actions:
   - You can issue a rollback for all of the work units.
   - You can terminate and let CMS end-of-command processing roll back all the work units.
   - You can determine (as described in "3" on page 310 above) whether the work unit(s) associated with the conversations must be backed out and then take the appropriate action.

Every APPC/VM protected conversation is associated with a work unit. A program can initiate multiple protected conversations on a work unit, but can accept only one.

Note that you cannot route a SYNCLVL=SYNCPT conversation through the TSAF virtual machine or a CS collection; only AVS machines and APPCVM programs communicating within the same system support protected conversations.

Scenario 3: Coordinating Resources

The following scenario shows how an assembler program can make updates to three files using a protected conversation.

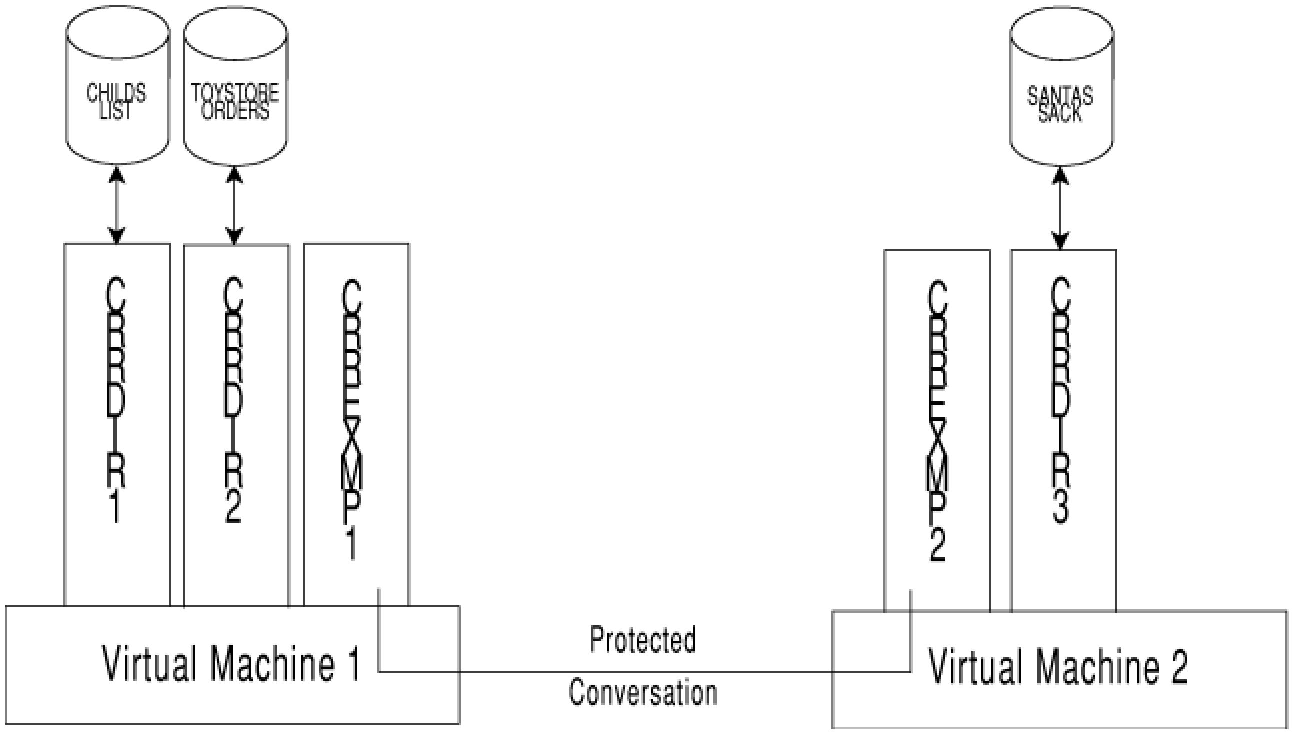


Figure 52. Updating Three SFS Files Using Coordinated Resource Recovery

The application, CRREXMP1, running in Virtual Machine 1, has access to two files, CHILDS LIST and TOYSTORE ORDERS. These files are located in two different SFS directories, CRRDIR1 and CRRDIR2, respectively. The application, CRREXMP2, running on Virtual Machine 2, has access to one file, SANTAS SACK, which is located in the SFS directory CRRDIR3.

Imagine that one virtual machine is operated by a child entering a Christmas list at a local toy store. The other virtual machine are handled by one of Santa's elves at the North Pole. As the child enters the toys, CRREXMP1 updates its two files, CHILDS LIST and TOYSTORE ORDER. CRREXMP1 then establishes a protected conversation to CRREXMP2 (at the North Pole). CRREXMP1 will send the list of toys to be added to the list Santa will bring the child (SANTA SACK). If CRREXMP2 is successful in updating its file (it will first have to see if the child has been naughty or nice), it will request that the updates be committed. Otherwise, it will request that the updates be rolled back.

Note: This same example is illustrated in *z/VM: CMS Application Development Guide* using CPI Communications.

Following is a high-level overview of the assembler code that could be used to complete this scenario. For information on the macros used in this scenario see:

- *z/VM: CMS Macros and Functions Reference* for CMSIUCV and HNDIUCV macros
- *z/VM: CP Programming Services* for APPCVM and IUCV macros in z/VM.

Sequence of Instructions for Updating Multiple Files

```
CRREXMP1                        CRREXMP2
   1  HNDIUCV SET                   1  HNDIUCV SET
   2  DMSSSPTO
   3  DMSSETAG
   4  DMSOPEN CHILDS LIST
   5  DMSWRITE CHILDS LIST
   6  DMSCLOSE CHILDS LIST
   7  DMSOPEN TOYSTORE ORDERS
```

```
        8  DMSWRITE TOYSTORE ORDERS
        9  DMSCLOSE TOYSTORE ORDERS
        10 APPCVM CONNECT SYNCLVL=SYNCPT, MF=L
        11 CMSIUCV CONNECT
                                 12 IUCV ACCEPT (MF=L)
                                 13 CMSIUCV ACCEPT
        14 APPCVM SENDCNF
                                 15 APPCVM RECEIVE
                                 16 APPCVM SENDCNFD
        17 APPCVM SENDDATA
        18 APPCVM RECEIVE TYPE=RECEIVE
                                 19 APPCVM RECEIVE
                                 20 APPCVM SENDCNFD

                                 21 DMSSPTO
                                 22 DMSSETAG

                                 23 DMSOPEN SANTAS SACK
                                 24 DMSWRITE SANTAS SACK
                                 25 DMSCLOSE SANTAS SACK

        26 CMSIUCV QCMSWID               26 CMSIUCV QCMSWID


 *************************************************************
 * One of three things can happen at this point:             *
 *  1. Both applications roll back the updates (steps 27-35).*
 *  2. Both applications commit the updates (steps 36-42).   *
 *  3. One application tries to commit the updates but the   *
 *     other one rolls back (steps 43-52).                   *
 *************************************************************



 Both CRREXMP1 and CRRECXP2 roll back updates
                                 27 DMSROLLB
         28 APPCVM RECEIVE
         29 DMSROLLB
                                 30 APPCVM SEVER MF=L
                                 31 CMSIUCV SEVER
         32 APPCVM RECEIVE
         33 APPCVM SEVER MF=L
         34 CMSIUCV SEVER
         35 HNDIUCV CLR                   35 HNDIUCV CLR

         Both CRREXMP1 and CRREXMP2 commit updates

                                  36 APPCVM SETMODFY TYPE=SEVER
                                  37 DMSCOMM
         38 APPCVM RECEIVE
         39 DMSCOMM
         40 APPCVM SEVER MF=L             40 APPCVM SEVER MF=L
         41 CMSIUCV SEVER                 41 CMSIUCV SEVER
         42 HNDIUCV CLR                   42 HNDIUCV CLR

      CRREXMP2 commits updates and then CRREXMP1 rolls back the updates

                                   43 APPCVM SETMODFY TYPE=SEVER
                                   44 DMSCOMM
         45 APPCVM RECEIVE
         46 DMSROLLB
                                   47 APPCVM SEVER MF=L
                                   48 CMSIUCV SEVER
         49 APPCVM RECEIVE
         50 APPCVM SEVER MF=L
         51 CMSIUCV SEVER
         52 HNDIUCV CLR                    52 HNDIUCV CLR
```

The following list explains the outline of instructions shown in "Sequence of Instructions for Updating Multiple Files" on page 311.

1. Both CRREXMP1 and CRREXMP2 assembler programs identify the program name to CMS with HNDIUCV SET.
2. DMSSSPTO sets the synchronization point options.
3. DMSSETAG sets the transaction tag.

4. DMSOPEN opens the CHILDS LIST file in the SFS directory, CRRDIR1.
5. DMSWRITE writes to the CHILDS LIST file.
6. DMSCLOSE with NOCOMMIT option closes the CHILDS LIST file.
7. DMSOPEN opens the TOYSTORE ORDERS file in the SFS directory, CRRDIR2.
8. DMSWRITE writes to the TOYSTORE ORDERS.
9. DMSCLOSE with NOCOMMIT option closes the TOYSTORE ORDERS file.
10. APPCVM CONNECT MF=L formats APPC/VM parameter list for connect.
11. CMSIUCV CONNECT connects to CRREXMP2 with the APPC/VM parameter list
12. IUCV ACCEPT MF=L formats the IUCV parameter list for ACCEPT.
13. CMSIUCV ACCEPT accepts CRREXMP1's connection.
14. APPCVM SENDCNF ensures that CRREXMP2 has received allocation.
15. APPCVM RECEIVE gets confirmation request.
16. APPCVM SENDCNFD responds to the confirmation request.
17. APPCVM SENDDATA sends the data record to CRREXMP2.
18. APPCVM RECEIVE TYPE=RECEIVE to confirm and enter receive state.
19. APPCVM RECEIVE receives the data record and CRREXMP's request to enter receive state.
20. APPCVM SENDCNFD confirms CRREXMP1's state request.
21. DMSSSPTO sets synchronization point options.
22. DMSSETAG sets the transaction tag.
23. DMSOPEN opens SANTAS SACK file in the SFS directory, CRRDIR3.
24. DMSWRITE writes to SANTAS SACK.
25. DMSCLOSE with NOCOMMIT option closes SANTAS SACK.
26. CMSIUCV QCMSWID determines workunitid for APPC/VM paths.*

```
**************************************************************
* One of three things can happen at this point:              *
*  1. Both applications roll back the updates (steps 27-35). *
*  2. Both applications commit the updates (steps 36-42).    *
*  3. One application tries to commit the updates but the    *
*     other one rolls back (steps 43-52).                    *
**************************************************************
```

Both CRREXMP1 and CRREXMP2 roll back updates

27. DMSROLLB initiates rollback processing for returned workunitid.
28. APPCVM RECEIVE receives the rollback indication.
29. DMSROLLB performs rollback processing for returned work.
30. APPCVM SEVER MF=L formats APPC/VM parameter list for sever abend.
31. CMSIUCV SEVER severs the protected conversation with the partner.
32. APPCVM RECEIVE gets the sever indication.
33. APPCVM SEVER MF=L formats the APPC/VM parameter list for normal sever
34. CMSIUCV SEVER severs local path for protected conversation.
35. Both CRREXMP1 and CRREXMP2 assembler programs notify CMS that the program is complete with HNDIUCV CLR.

Both CRREXMP1 and CRREXMP2 commit updates

36. APPCVM SETMODFY TYPE=SEVER severs the conversation after a successful sync point.
37. DMSCOMM initiates commit processing for the returned workunitid.
38. APPCVM RECEIVE receives the commit request indication.

39. DMSCOMM performs commit processing for returned workunitid.
40. APPCVM SEVER MF=L formats the APPC/VM parameter list for normal sever.
41. CMSIUCV SEVER severs the local path for the protected conversation.
42. Both CRREXMP1 and CRREXMP2 assembler programs notify CMS that the program is complete with HNDIUCV CLR.

CRREXMP2 commits the updates and then CRREXMP1 rolls back the updates

43. APPCVM SETMODFY TYPE=SEVER severs the conversation after a successful sync point.
44. DMSCOMM initiates commit processing for returned workunitid.
45. APPCVM RECEIVE receives the commit request indication.
46. DMSROLLB performs rollback processing for returned workunitid. This causes the updates made by CRREXMP1 and CRREXMP2 to be rolled back.
47. APPCVM SEVER MF=L formats the APPC/VM parameter list for sever abend.
48. CMSIUCV SEVER severs the protected conversation with CRREXMP1.
49. APPCVM RECEIVE gets the sever indication.
50. APPCVM SEVER MF=L formats the APPCVM parameter list for normal sever.
51. CMSIUCV SEVER severs the local path for the protected conversation.
52. Both CRREXMP1 and CRREXMP2 assembler programs notify CMS that the program is complete with HNDIUCV CLR.

Part 5. OS/MVS Simulation

This part of the document describes how CMS simulates OS/MVS services. Part 5, " OS/MVS Simulation," on page 315 includes the following chapters:

- Chapter 22, "Developing OS/MVS Programs under CMS," on page 317 describes how to develop OS/MVS programs that can be simulated in CMS.
- Chapter 23, "Using OS/MVS Simulated Data Sets in CMS," on page 349 describes how to use OS/MVS simulated data sets in CMS.

Chapter 22. Developing OS/MVS Programs under CMS

This chapter describes:

- Background of OS/MVS Simulation
- Considerations for using OS/MVS macros in CMS
- OS/MVS macros that CMS simulates
- OS/MVS Resource Management
- Using CMS libraries to aid in OS/MVS program development
- OS/MVS and CMS terminology.

For more information on OS/MVS simulation, see Chapter 23, "Using OS/MVS Simulated Data Sets in CMS," on page 349.

Note: The programming interfaces defined by the MVS operating system and simulated by CMS are documented in this chapter and in Chapter 23, "Using OS/MVS Simulated Data Sets in CMS," on page 349. For information about these interfaces, see the MVS documentation.

OS/MVS Simulation History

When the VM operating system was being developed, applications and licensed programming products were already running under the OS/VS1 or DOS operating systems. The VM designers saw the need to allow these applications and program products to run in the new VM environment where our customers could use them for new program development and testing, a major task for almost every customer shop. Because the available IBM licensed products had to run in both operating environments, we needed to develop a subsystem which would allow them to use the OS interface on VM. The CMS OS Simulation subsystem was developed to satisfy that requirement.

Today, many basic OS/MVS functions are supported using the CMS OS Simulation subsystem. You should note that it does not support all of either the OS/MVS macros or the SVC functions. The subsystem was intended to support just those functions needed to allow the IBM program products and featured applications to run under VM/CMS with little or no modification of the program products themselves. A list of the OS/MVS macros and SVC functions that CMS supports can be found in Table 28 on page 320, Table 29 on page 328, Table 30 on page 335, and Table 31 on page 335.

How CMS Performs OS/MVS Simulation

When CMS OS Simulation is used, VM does not invoke MVS to handle the function. Instead, it calls a series of programs within the CMS operating system to simulate the function of the OS/MVS system. This simulation is not exact because OS Simulation calls CMS native functions to provide the actual services.

CMS tries to functionally simulate OS/MVS macro and SVC function interfaces in a way which provides results to OS programs executing under CMS equivalent to the results obtained running on an MVS system. MVS macros and SVC functions expand to call CMS OS Simulation routines which do setup, provide control block and device interface information, and/or invoke CMS native support routines to perform the required functions. At the same time, OS Simulation maintains pseudo-control blocks for the OS/MVS products so that they can query and manipulate data as if they were running in an OS/MVS type environment.

Programs using simulated OS/MVS macros run in an environment defined by CMS commands. OS Simulation runs on top of the CMS virtual machine configuration. The accessed disks and directories and the files residing on them comprise the files which OS Simulation will use. The FILEDEF and GLOBAL commands define the files that the particular program will use and associate them with particular DCBs the same way the data definition (DD) statements in job control language (JCL) define the datasets that the program would use if it ran in the MVS batch environment.

Using OS/MVS Macros in CMS Programs — Some Considerations

Before you use OS/MVS macros in your programs, you should note the following considerations:

1. OS Simulation becomes active when the first DCB is opened. It becomes inactive when the last DCB is closed. While OS Simulation is active, no CMS commands should be issued which will change the OS Simulation environment for any open DCB. It is best not to issue any CMS commands or macros that could change the OS Simulation environment while OS Simulation is active.

   The list of commands and macros which would change the OS Simulation environment begins with the GLOBAL, FILEDEF, and LABELDEF commands which directly define the OS Simulation environment. The list extends to the CMS file system commands and macros which define the files in the OS Simulation environment. These include the FSOPEN, FINIS, FSWRITE, and FSREAD I/O macros, the COPY, RENAME, and ERASE commands which create or delete files, the LINK, DETACH, ACCESS, RELEASE, and FORMAT commands which manipulate disks and directories, and the TAPE command which sets up tapes. Issuing any of these commands while OS Simulation is active changes the environment and can cause unpredictable errors such as abending CMS.
2. Most, but not all, OS/MVS macros are supported. Some of these are supported for execution under CMS. Others exist only to allow programs to be assembled or compiled under CMS for development. The programs which use these restricted macros must actually be executed in a true OS/MVS environment.
3. The simulation of OS/MVS macros by CMS is not necessarily the same as the current MVS™ support. CMS simulates only a selected subset of OS/MVS macros and, because of operational differences between VM and MVS, macros that are supported may work differently between the two systems.
4. Not all parameters are simulated on the OS/MVS macros CMS supports. Since all parameters do assemble, it is very important to note which parameters are simulated. CMS ignores parameters it does not simulate and no error messages are supplied. See "OS/MVS Macros That CMS Simulates" on page 319 for a list of the macros CMS supports and the parameters CMS simulates.
5. The OS/MVS macros in CMS provide an OS/MVS interface to CMS services. If you are developing a program exclusively for CMS, you should use CMS macros rather than OS/MVS macros.
6. MVS/XA supports System/370 and 370-XA expansions for most, **but not all**, macros. Various MVS macros (such as WTO, ESTAE, and ATTACH) require you to use the SPLEVEL macro to generate the proper expansion. If you use these macros in an application you plan to run on MVS/XA, you must use SPLEVEL to generate the proper expansion. See the *MVS/XA Conversion Notebook* for information on how and when MVS/XA requires you to use the SPLEVEL macro.

   Notes:

   a. Only CMS levels prior to CMS Level 12 can execute in a 370 virtual machine.
   b. If you do not plan to run your application on MVS, you do not need to use the SPLEVEL macro. CMS supports System/370, 370-XA, and ESA/XC expansions for all the OS/MVS macros it simulates; when you code an OS/MVS macro for a program you run on CMS, you do not need to worry about what mode expansion gets generated.
7. OS/MVS macros simulated by CMS are not access-register mode (AR mode) capable. The calling program must ensure that it is not in AR mode when the macro call is issued; otherwise CMS OS/MVS simulation support will terminate the call with aX'0F8' abend code and reason code X'18'. This support is consistent with current MVS/ESA* support.

   Access registers are saved across the interface call and are restored upon return to the caller. The CMS OS/MVS simulation function called by the interface executes in primary-space mode (non-AR mode), so parameters associated with the call must refer to data in the caller's primary-address space or unpredictable results may occur.
8. CMS OS Simulation is not an OS or MVS multitasked environment. It is by design a single user, synchronous environment where the calling program waits until subtasks complete their function and

return. In many cases this means the user session is 'tied up' until the requested simulation function completes. These subtasks may require that an interactive environment be maintained to allow user to respond to OS Simulation subsystem prompts.

9. Several restrictions still apply for AMODE and RMODE execution of these simulated services. Most of the control blocks and interface parameter lists are designed to the original OS/MVS conventions of 24-bit addressing. Frequently the address fields contain control or flag information in the high order byte. This requires that many of the key control blocks, which may be defined by user applications, reside below the 16M line to allow proper address resolution. Applications must also ensure that OS Simulation services are called while in the proper AMODE to ensure addressability.
10. Storage resource management can be done using either OS/MVS macro calls (GETMAIN,FREEMAIN) or CMS native services (CMSSTOR). Both are actually implemented via the same CMS storage management subsystem. However, the default subpool name in which CMSSTOR obtains storage is USER, while the default subpool name used by GETMAIN/FREEMAIN is DMSOS000. (Note: The subpool name used by GETMAIN/FREEMAIN can be changed to DMSOS*nnn* by specifying the SP= parameter, where *nnn* is the value specified on the SP= parameter.) Because GETMAIN automatic storage cleanup can be affected by the CMS STORECLR setting, it is not advised to mix these services.

OS/MVS Macro Libraries

There are several levels and usage classification of macro support for OS/MVS functions which CMS maintains. Four major macro libraries differentiate these levels:

OSMACRO
Contains the macros that CMS provides for execution of programs using MVS interfaces in 370, XA, or XC virtual machines.

Note: Only CMS levels prior to CMS Level 12 can execute in a 370 virtual machine.

MVSXA
Contains the simulated MVS/XA versions of the OS/MVS macros for the execution of programs using MVS interfaces in XA or XC virtual machines. These macros expand to allow 31-bit addressing exploitation.

OSMACRO1
Contains the nonsimulated versions of OS/MVS macros that are used only for assembly only under CMS.

OSVSAM
Contains the subset of supported OS/VSAM macros.

Note: The MACLIBs listed previously contain MVS SP 2.2.0 macros and DFP 2.3.0 macros.

Refer to "Using Macro Libraries" on page 21 for information on CMS and CP macro libraries.

OS/MVS Macros That CMS Simulates

When a language processor or a user-written program is executing in the CMS environment and using OS-type functions, it is not executing OS/MVS code. Instead, CMS provides routines that simulate the OS/MVS functions required to support OS/MVS language processors and their generated object code.

CMS functionally simulates the OS/MVS macros in a way that presents equivalent results to programs executing under CMS. The OS/MVS macros are supported only to the extent stated in the publications for the supported language processors and then only to the extent necessary to successfully satisfy the specific requirement of the supervisory function.

For more information on the data management macros, see the *MVS/XA Data Administration: Macro Instruction Reference* and the *MVS/XA Data Administration Guide*. For information on the supervisor macros, see *MVS/XA Supervisor Services and Macro Instructions*.

The tables list the OS/MVS macros that CMS supports.

Notes:

1. The following charts list only the parameters that CMS functionally supports for each macro. The MVS versions of these macros may include other parameters that CMS ignores. Parameters that CMS ignores assemble correctly; they just don't do anything.
2. For some parameters, CMS supports only certain options. For example, CMS supports XCTL=NO for STAE/ESTAE and DEFER=NO for STAX; it ignores XCTL=YES for STAE/ESTAE and DEFER=YES for STAX. Because CMS does not flag ignored parameters during assembly, you should carefully check the results of your program.
3. Some parameters such as PURGE for STAE/ESTAE and some macros such as ATTACH have no CMS equivalents; they assemble correctly but are ignored during execution.

MVS/XA Data Management Macros

Table 28. MVS/XA Data Management Macros That CMS Simulates

Macro	Description and Parameters/Options Supported
BLDL	Builds a directory list for a partitioned data set. *dcb_address* , *list_address*
BSP	Backs up a record on a tape or disk. *dcb_address*
BUILDRCD	Causes a buffer pool and a record area to be constructed. *area_address* , *number_of_buffers* , *buffer_length* , *record_area_address* , *record_area_length*
CHECK	Verifies READ/WRITE completion. *decb_address* , DSORG = ALL
CHKPT	No-op. NOP
CLOSE	Completes and secures I/O processing on a DCB. (*dcb_address*) , REREAD , LEAVE , REWIND , TYPE = T
CNTRL	No-op. NOP

Table 28. MVS/XA Data Management Macros That CMS Simulates (continued)

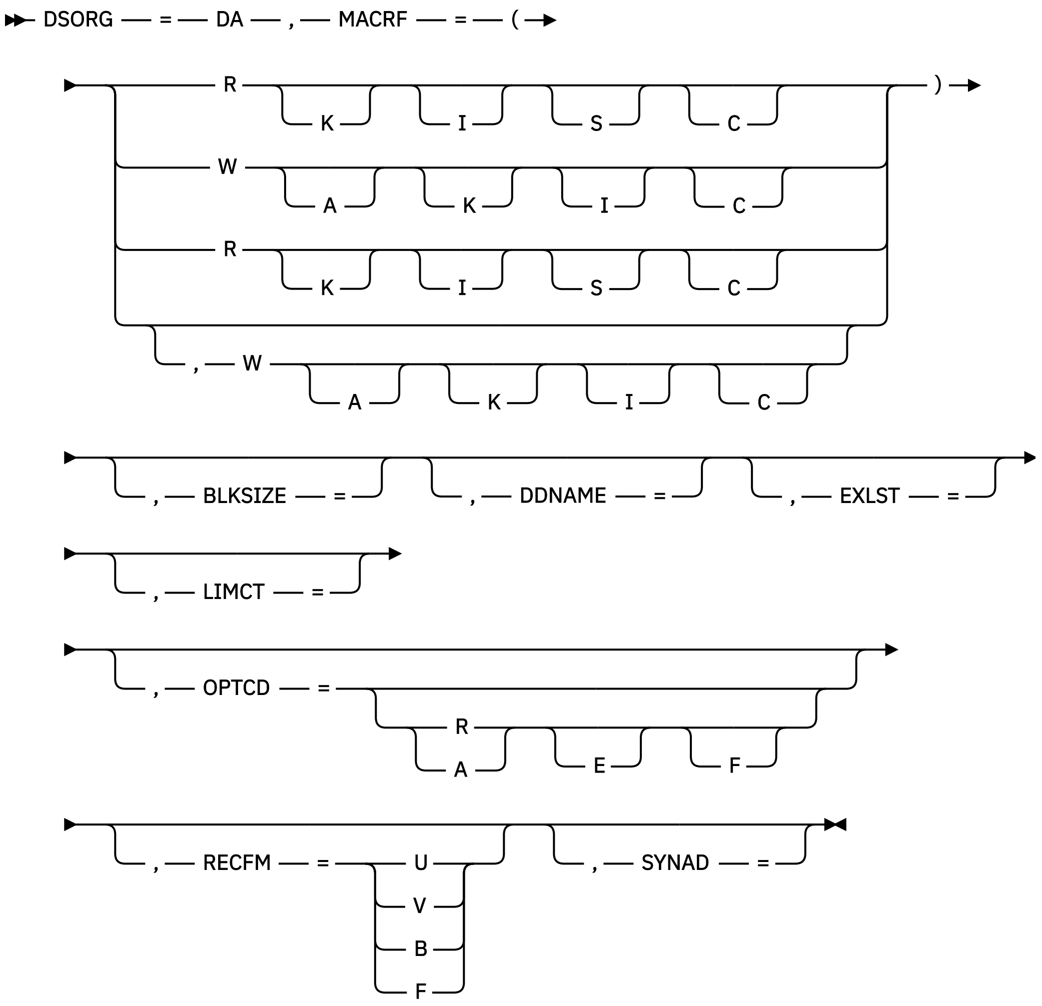
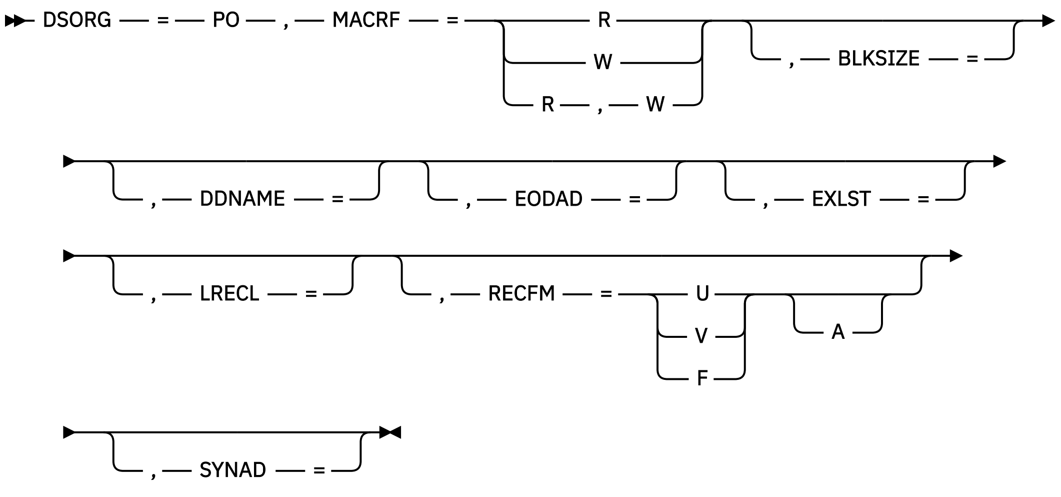
Macro	Description and Parameters/Options Supported
DCB (BDAM)[13]	Constructs a data control block for BDAM.  DSORG = DA , MACRF = (R K I S C W A K I C R K I S C , W A K I C) , BLKSIZE = , DDNAME = , EXLST = , LIMCT = , OPTCD = R A E F , RECFM = U V B F , SYNAD =
DCB (BPAM)[13]	Constructs a data control block (BPAM).  DSORG = PO , MACRF = R W R , W , BLKSIZE = , DDNAME = , EODAD = , EXLST = , LRECL = , RECFM = U V F A , SYNAD =

Table 28. MVS/XA Data Management Macros That CMS Simulates (continued)

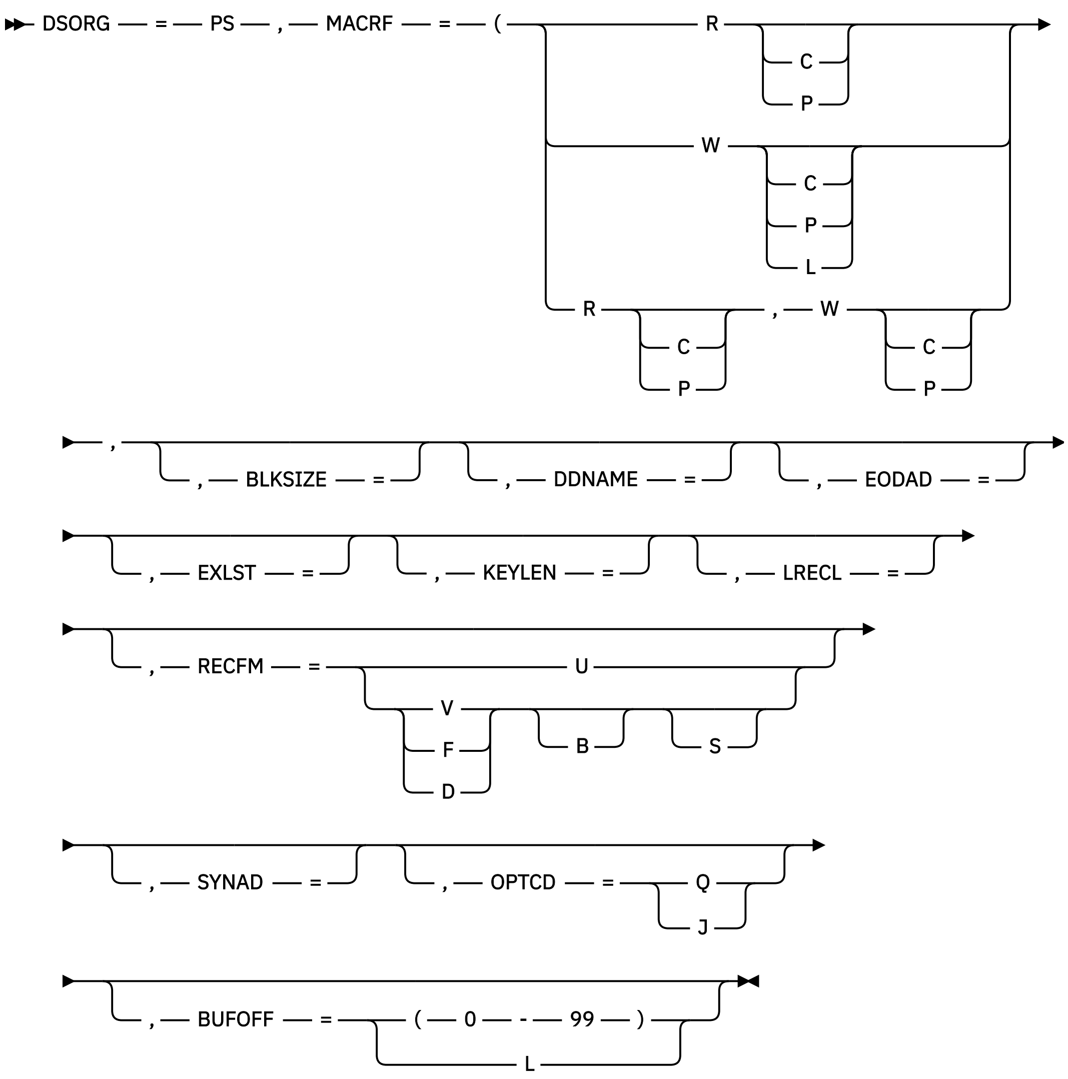
Macro	Description and Parameters/Options Supported
DCB (BSAM)[13]	Constructs a data control block (BSAM).  DSORG = PS , MACRF = (R C P W C P L R C P , W C P) , , BLKSIZE = , DDNAME = , EODAD = , EXLST = , KEYLEN = , LRECL = , RECFM = U V F D B S , SYNAD = , OPTCD = Q J , BUFOFF = (0 - 99) L

Table 28. MVS/XA Data Management Macros That CMS Simulates (continued)

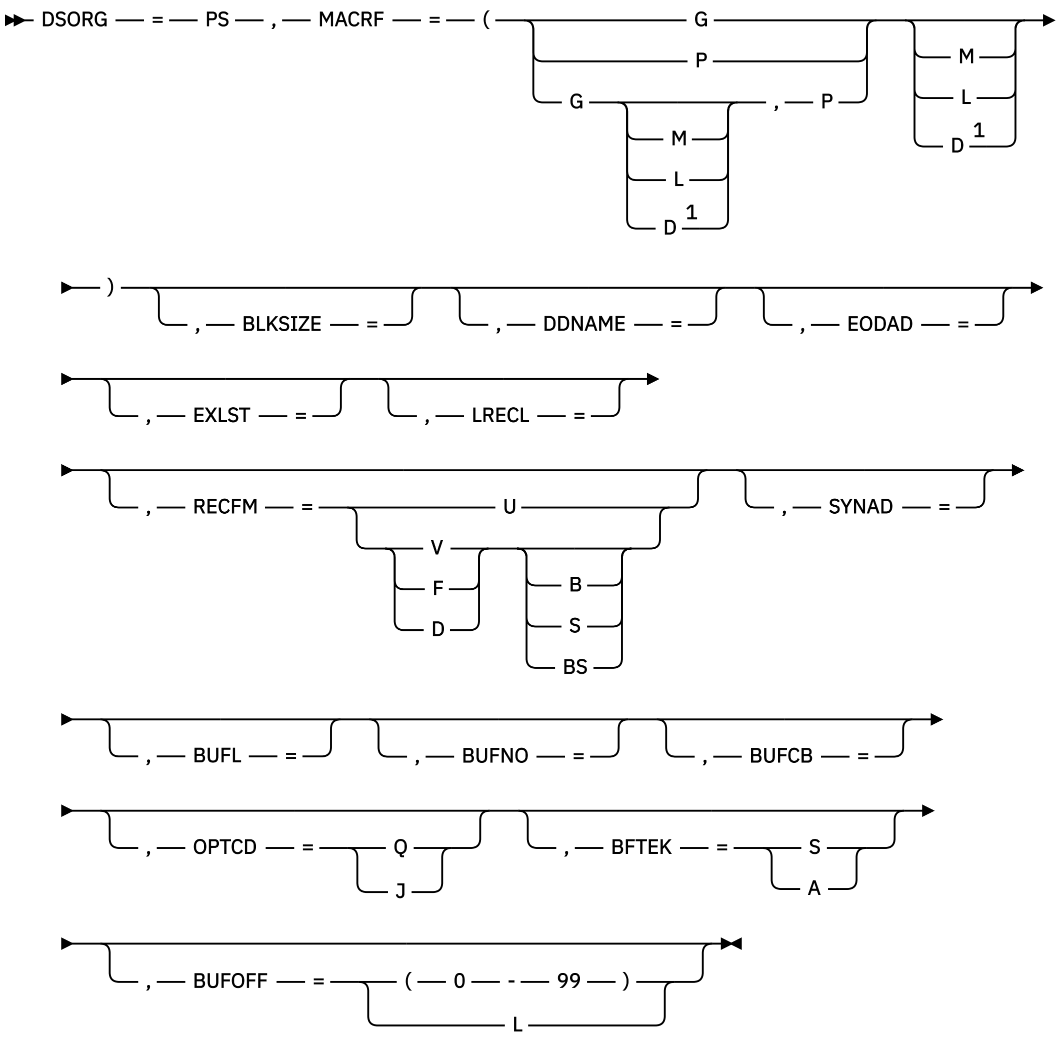
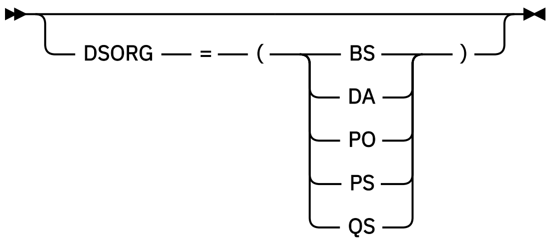
Macro	**Description and Parameters/Options Supported**
DCB (QSAM)[13]	Constructs a data control block (QSAM).  DSORG = PS , MACRF = (G P G M L D 1 , P M L D 1) , BLKSIZE = , DDNAME = , EODAD = , EXLST = , LRECL = , RECFM = U V F D B S BS , SYNAD = , BUFL = , BUFNO = , BUFCB = , OPTCD = Q J , BFTEK = S A , BUFOFF = (0 - 99) L Notes: [1] MACRF of D is supported for assembly only. Data mode is not supported for execution.
DCBD	Generates a DSECT for a data control block. 

Table 28. MVS/XA Data Management Macros That CMS Simulates (continued)

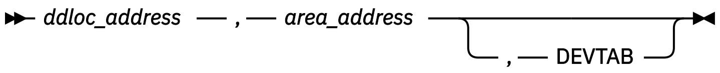
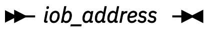
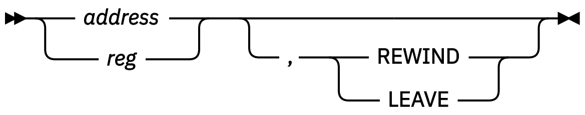
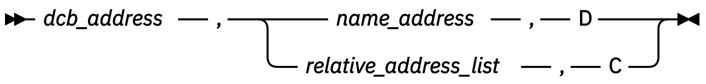
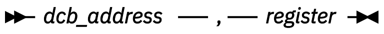
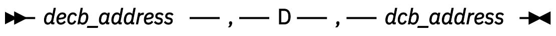
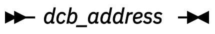
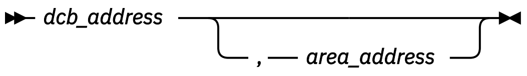
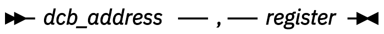
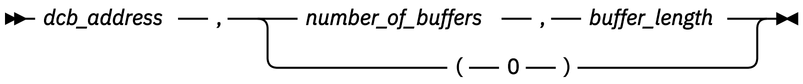
Macro	Description and Parameters/Options Supported
DEVTYPE	Obtains device-type physical characteristics.  **Note:** The DEVTYPE interface will not return valid track or cylinder details that can be used for DASD space calculations. It is intended only to give access to default device characteristics. If detailed real DASD device characteristics are needed, see CP DIAGNOSE X'210' in the *z/VM: CP Programming Services* or the CMS 'DEVTYPE' utility program in the *z/VM: General Information*.
EXCP	Executes a channel program for graphic access method (GAM). 
FEOV	Forces an EOV condition on a tape or DASD file. 
FIND	Locates a member of a partitioned data set. 
FREEBUF	Returns a buffer to the DCB buffer pool. 
FREEDBUF	Releases a simulated BDAM buffer. 
FREEPOOL	Releases the DCB buffer pool. 
GET	Reads system-blocked data (QSAM). GET is supported for Locate and Move modes only. 
GETBUF	Acquires DCB buffer storage. 
GETPOOL	Constructs a buffer pool for a DCB. 

Table 28. MVS/XA Data Management Macros That CMS Simulates (continued)

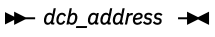
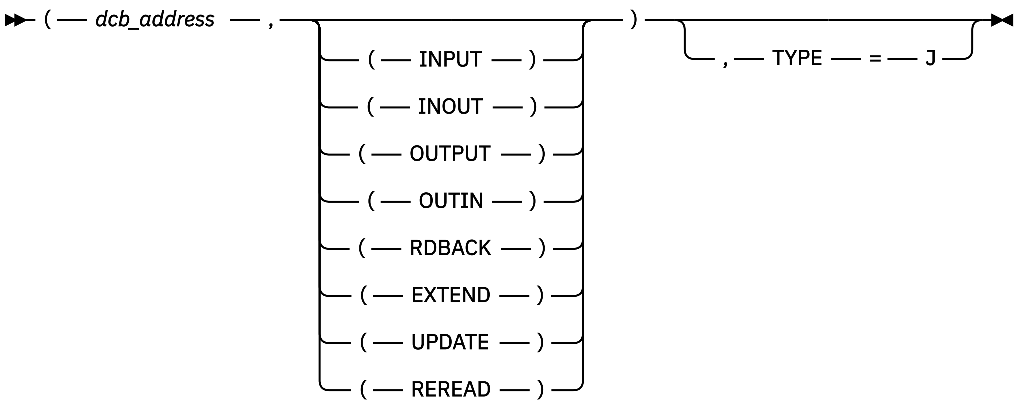
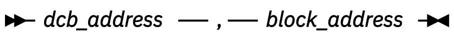
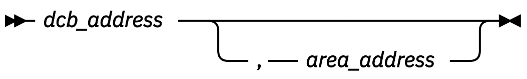
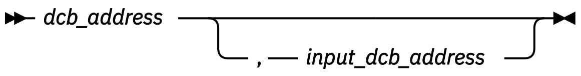
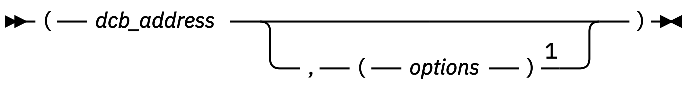
Macro	Description and Parameters/Options Supported
NOTE	Manages data set positioning. 
OPEN	Prepares a DCB for I/O processing.  (dcb_address , (INPUT) (INOUT) (OUTPUT) (OUTIN) (RDBACK) (EXTEND) (UPDATE) (REREAD)) , TYPE = J
POINT	Manages data set positioning. 
PUT	Writes system-blocked data (QSAM). PUT is supported for Locate and Move modes only. 
PUTX	Returns the updated record to the data set from which it was read (QSAM). 
RDJFCB	Obtains information from FILEDEF command about an OS/MVS data set.  Notes: [1] RDJFCB has the same options as OPEN.

Table 28. MVS/XA Data Management Macros That CMS Simulates (continued)

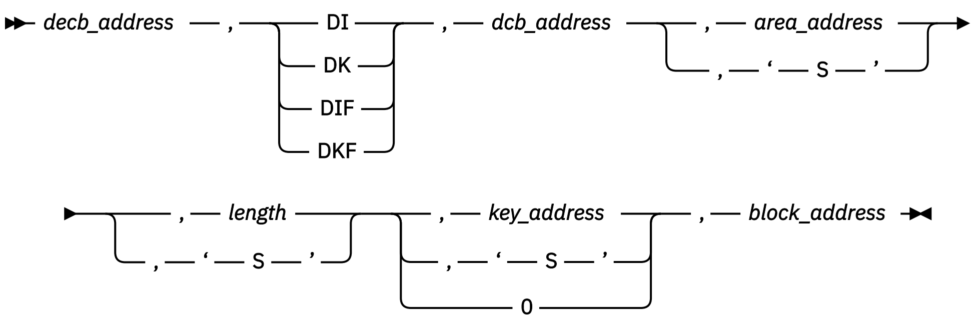
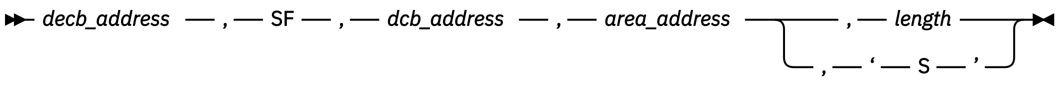
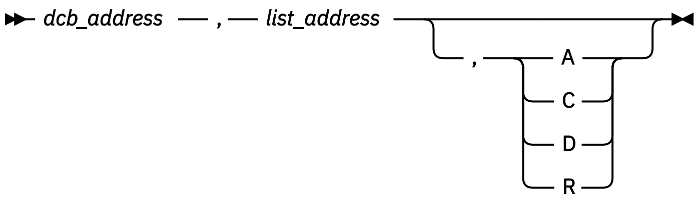
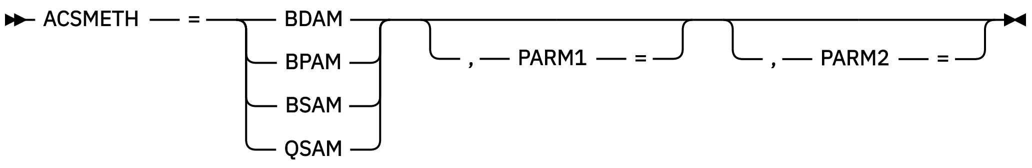
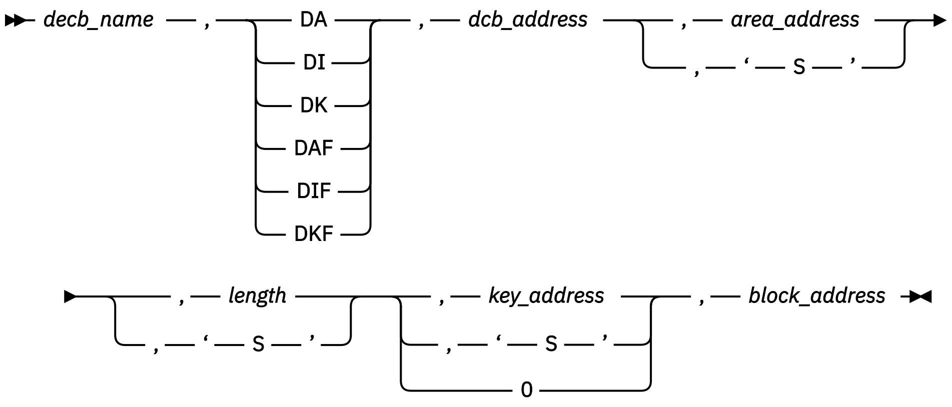
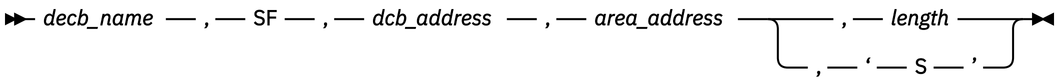
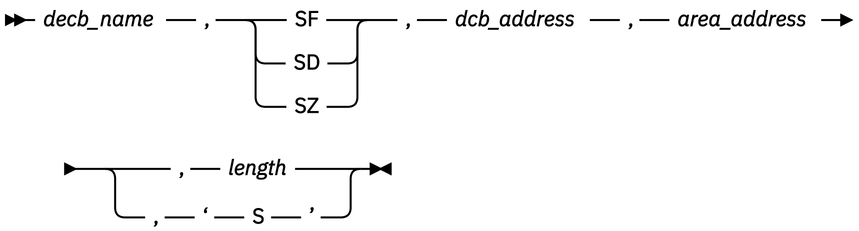
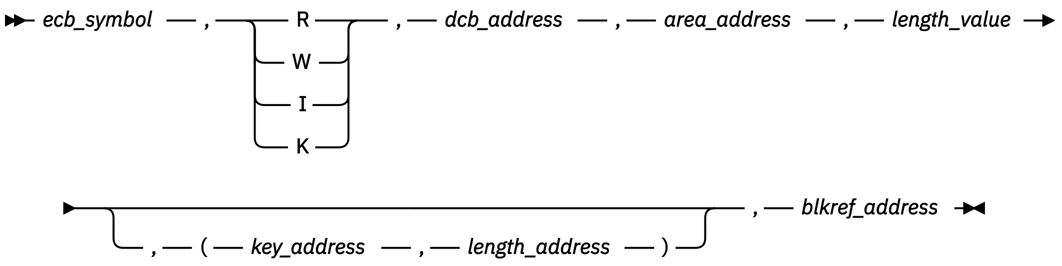
Macro	Description and Parameters/Options Supported
READ (BDAM)	Reads a physical input record (BDAM).  decb_address , DI DK DIF DKF , dcb_address , area_address , ‘ S ’ , length , ‘ S ’ , key_address , ‘ S ’ 0 , block_address
READ (BPAM and BSAM)	Reads a physical input record (BSAM, BPAM).  decb_address , SF , dcb_address , area_address , length , ‘ S ’
RELSE	No-op. NOP
STOW	Updates partitioned dataset directories. 
SYNADAF	Provides SYNAD analysis function.  ACSMETH = BDAM BPAM BSAM QSAM , PARM1 = , PARM2 =
SYNADRLS	Releases SYNADAF message and save areas. *(no parameters)*

Table 28. MVS/XA Data Management Macros That CMS Simulates (continued)

Macro	Description and Parameters/Options Supported
WRITE (BDAM)	Writes a physical record (BDAM).  decb_name , DA DI DK DAF DIF DKF , dcb_address , area_address , 'S' , length , 'S' , key_address , 'S' 0 , block_address
WRITE (BPAM and BSAM)	Writes a physical record (BSAM, BPAM). 
WRITE (Create BDAM Data Set With BSAM)	Writes a physical record (BSAM, BDAM).  decb_name , SF SD SZ , dcb_address , area_address , length , 'S'
XDAP	Reads or writes direct access volumes.  ecb_symbol , R W I K , dcb_address , area_address , length_value , (key_address , length_address) , blkref_address

[13] CMS supports DCBs only below the 16MB line and in 24-bit addressing mode only.

MVS/XA Supervisor Macros

Table 29. MVS/XA Supervisor Macros That CMS Simulates

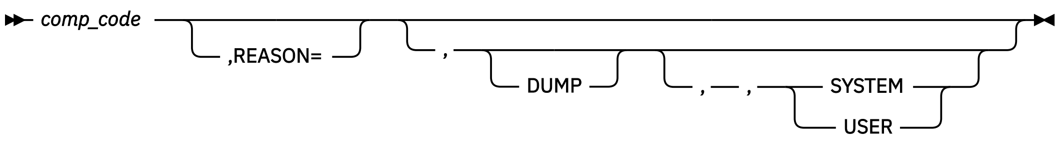
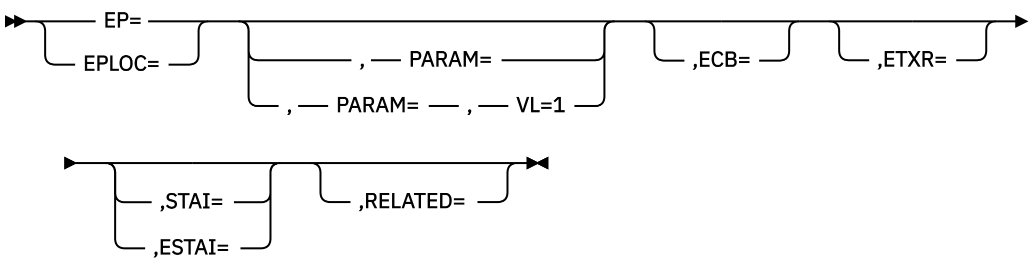
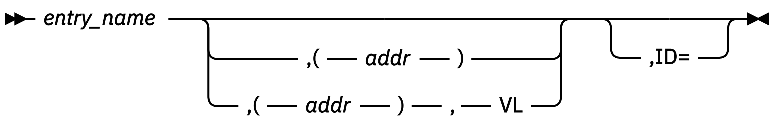
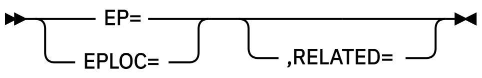
Macro	Description and Parameters/Options Supported
ABEND	Terminates processing with user-specified completion and reason codes. 
ATTACH	Passes control to another program at a new task level  EP= EPLOC= , PARAM= , PARAM= , VL=1 ,ECB= ,ETXR= ,STAI= ,ESTAI= ,RELATED=
CALL	Transfers control to a control section at a specified entry. 
CHAP	No-op. NOP
DELETE	Deletes a loaded program. 
DEQ	No-op. NOP
DETACH	No-op. NOP
ENQ	No-op. NOP

Table 29. MVS/XA Supervisor Macros That CMS Simulates (continued)

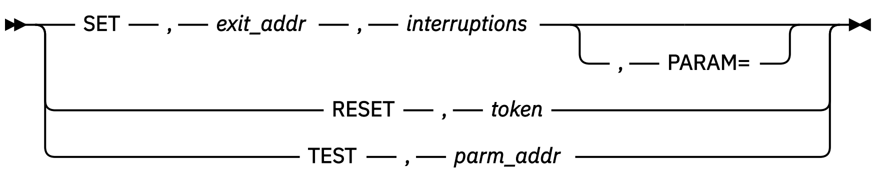
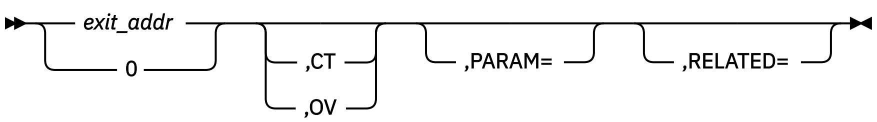
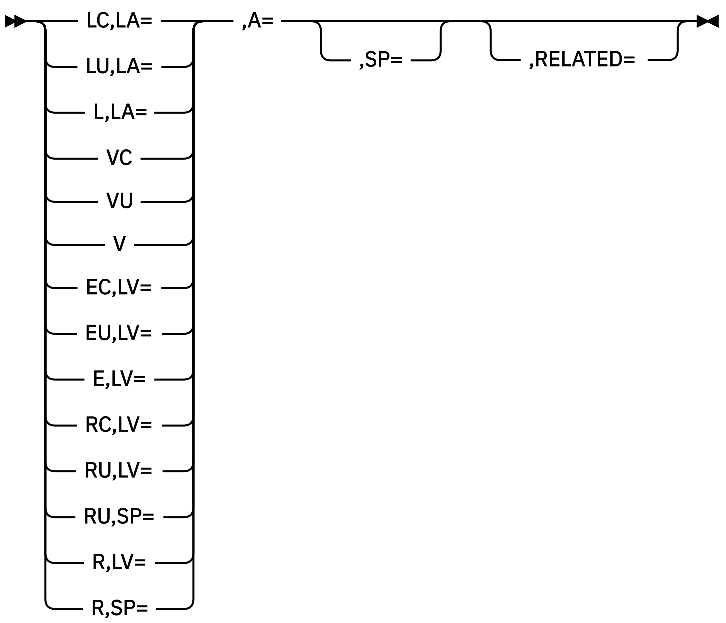
Macro	Description and Parameters/Options Supported
ESPIE	Sets up handlers for program interrupts in XA and XC mode.  SET , exit_addr , interruptions , PARAM= RESET , token TEST , parm_addr
ESTAE	Sets up abend exit routines in 370, XA, and XC virtual machines. 
EXTRACT	No-op. NOP
FREEMAIN	Releases user-acquired storage.  LC,LA= LU,LA= L,LA= VC VU V EC,LV= EU,LV= E,LV= RC,LV= RU,LV= RU,SP= R,LV= R,SP= ,A= ,SP= ,RELATED=

Table 29. MVS/XA Supervisor Macros That CMS Simulates (continued)

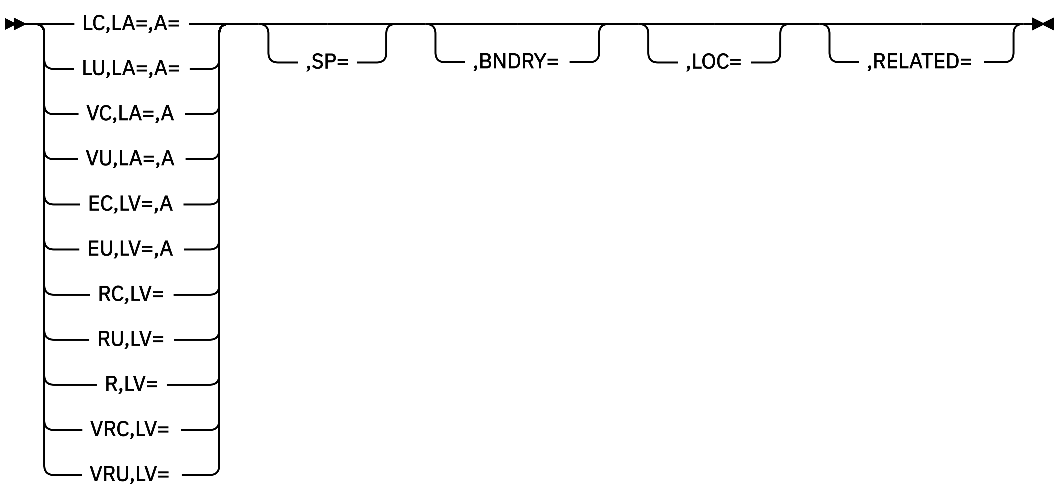
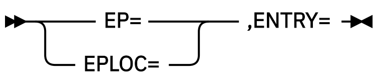
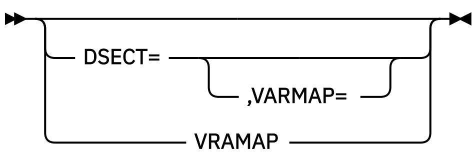
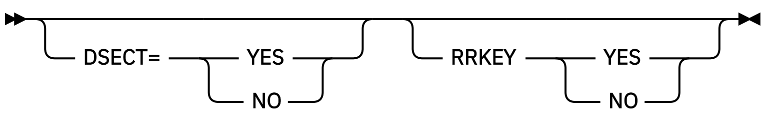
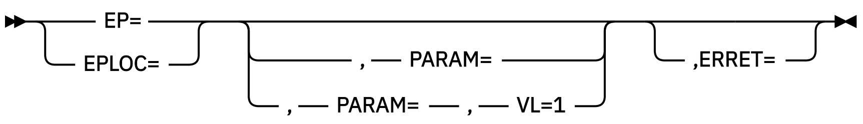
Macro	Description and Parameters/Options Supported
GETMAIN	Acquires user storage.  LC,LA=,A= LU,LA=,A= VC,LA=,A VU,LA=,A EC,LV=,A EU,LV=,A RC,LV= RU,LV= R,LV= VRC,LV= VRU,LV= ,SP= ,BNDRY= ,LOC= ,RELATED=
IDENTIFY	Adds an entry name to a loaded program. 
IHAEPIE	EPIE work area mapping macro. *(no parameters)*
IHASDWA	Mapping macro for the system diagnostic work area used in ESTAE.  **Note:** On entry to an ESTAE routine, SDWAGR1 and SDWAGR15 contain abend information and not the contents of the registers at the time of the abend.
IHAVRA	Mapping macro for the system diagnostic work area variable recording area. 
LINK	Passes control to another program at the same task level and returns to the calling program. 

Table 29. MVS/XA Supervisor Macros That CMS Simulates (continued)

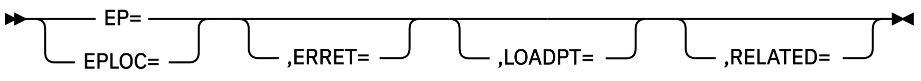
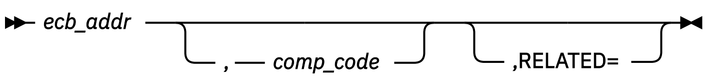
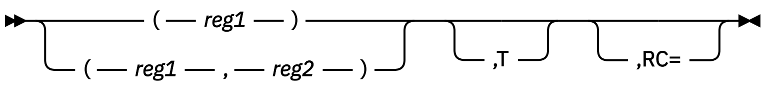
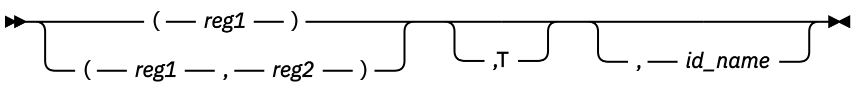
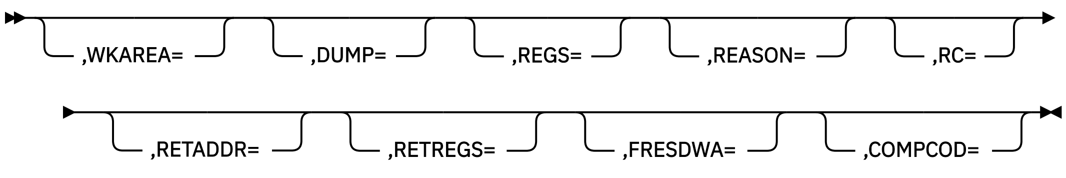
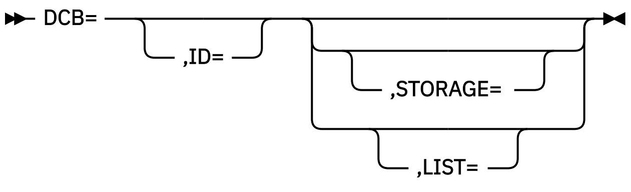
Macro	Description and Parameters/Options Supported
LOAD	Reads a program into storage. 
PGLOAD	No-op. NOP
PGOUT	No-op. NOP
PGRLSE	No-op. NOP
PGSER	No-op. NOP
POST	Signals event completion. 
RETURN	Returns from a called program. 
SAVE	Saves program registers. 
SETRP	Makes requests for recovery from an ESTAE/ESTAI exit.  ,WKAREA= ,DUMP= ,REGS= ,REASON= ,RC= ,RETADDR= ,RETREGS= ,FRESDWA= ,COMPCOD=
SNAP	Dumps specified areas of storage. 

Table 29. MVS/XA Supervisor Macros That CMS Simulates (continued)

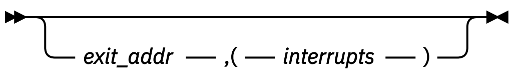
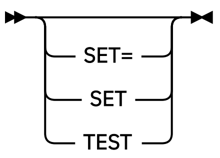
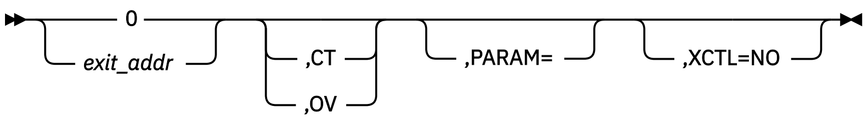
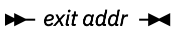
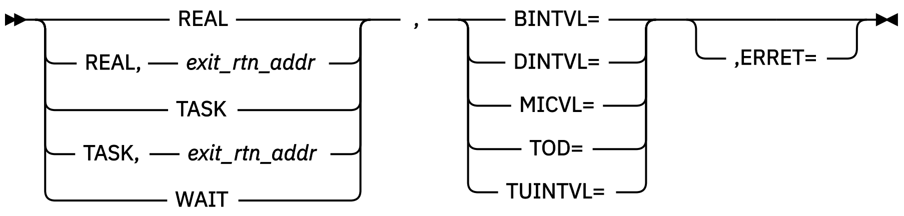
Macro	**Description and Parameters/Options Supported**
SPIE	Sets up an exit to be given control under user selected program interrupts. 
SPLEVEL	Sets System/370 or 370-XA macro expansion. 
STAE	Sets up an abend exit routine in a 370 virtual machine. 
STAX	Sets or cancels user exit for terminal attention interrupts 
STIMER	Sets the timer interval and the timer exit routine.  REAL REAL, *exit_rtn_addr* TASK TASK, *exit_rtn_addr* WAIT , BINTVL= DINTVL= MICVL= TOD= TUINTVL= ,ERRET=

Table 29. MVS/XA Supervisor Macros That CMS Simulates (continued)

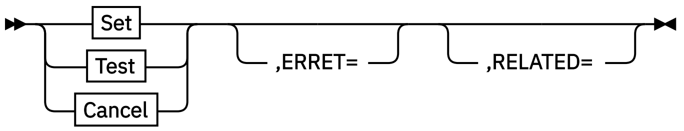
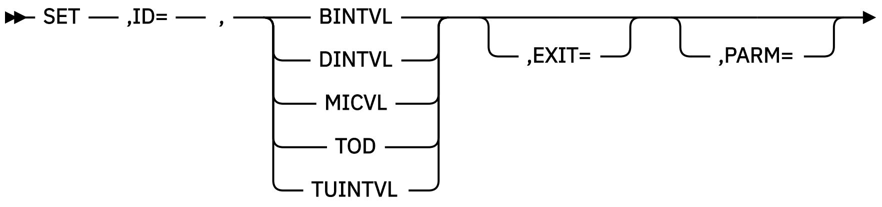
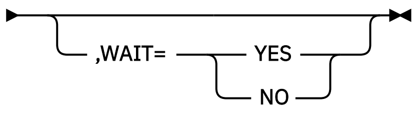
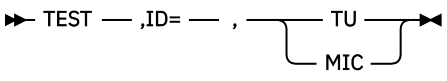
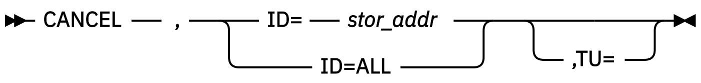
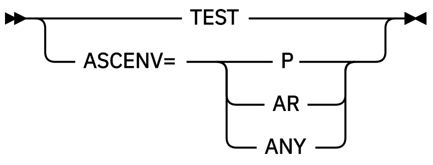
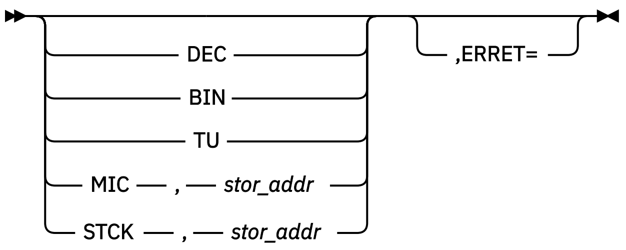
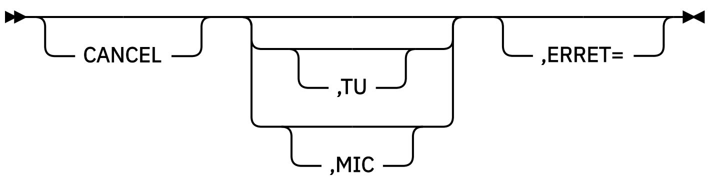
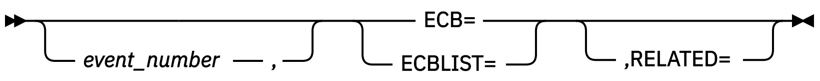
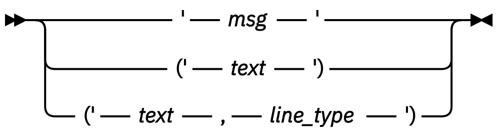
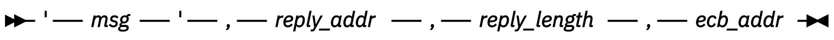
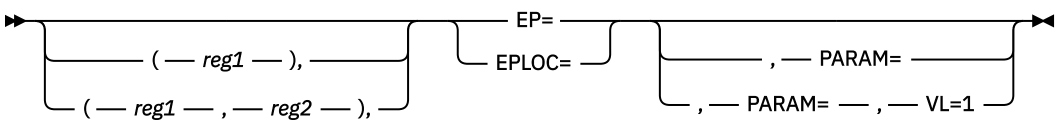
Macro	Description and Parameters/Options Supported
STIMERM	Sets, tests, or cancels multiple timer intervals and the timer exit routines.  **Set**  SET ,ID= , BINTVL DINTVL MICVL TOD TUINTVL ,EXIT= ,PARM=  **Test**  **Cancel** 
SYSSTATE	Conditions preferred-group macros so that access-register mode toleration code is expanded at assembly time. 
TIME	Gets the time of day.  DEC BIN TU MIC , stor_addr STCK , stor_addr ,ERRET=

Table 29. MVS/XA Supervisor Macros That CMS Simulates (continued)

Macro	Description and Parameters/Options Supported
TTIMER	Tests or cancels the timer. 
WAIT	Waits for one or more events. 
WTO	Writes a message to the operator's terminal. 
WTOR	Writes a message to the operator's terminal and requests a reply. 
XCTL	Passes control to another program at the same task level and does not return to the calling program.  (reg1), (reg1 , reg2), EP= EPLOC= , PARAM= , PARAM= , VL=1

Notes:

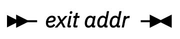
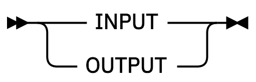
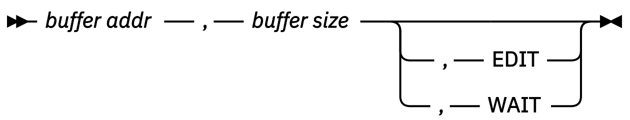
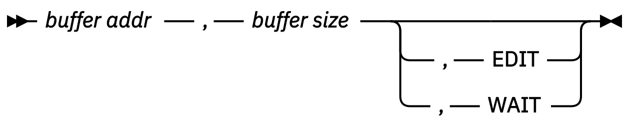
1. Complete documentation on MVS macros can be found in the MVS documentation. See the beginning of this chapter for more information on some limitations and considerations for using OS/MVS macros in CMS programs.
2. All of the macros in Table 28 on page 320 and Table 29 on page 328 are contained in OSMACRO MACLIB and MVSXA MACLIB.
   - OSMACRO MACLIB contains the macros that CMS provides for execution of programs using MVS interfaces in 370, XA, or XC virtual machines.

     Note: Only CMS levels prior to CMS Level 12 can execute in a 370 virtual machine.
   - MVSXA MACLIB contains the simulated MVS/XA versions of the OS/MVS macros for the execution of programs using MVS interfaces in XC or XA virtual machines.
3. With the following exceptions, all macros listed in Table 28 on page 320 and Table 29 on page 328 run in XC, XA, and 370 virtual machines:
   - ESPIE works in XC and XA virtual machines only.
   - SPIE, STAE, and the STAI parameter of the ATTACH macro work in 24-bit addressing mode only.

  - In an XC or XA virtual machine, DCBs must reside below the 16MB line. (The same restriction applies in MVS/XA.)
  - MVS/XA supports 370-XA and System/370 expansions for most, but not all, macros. For details, refer to the sixth consideration under "Using OS/MVS Macros in CMS Programs — Some Considerations" on page 318.
4. Several of the macros are no-ops (NOPs). They are included for portability purposes and pass control to CMS using an SVC instruction or a direct branch. CMS returns control to the program.
5. CLOSE TYPE=T (TCLOSE) is supported `only` for tape and DASD devices. CMS ignores a TCLOSE to any other device and processes the next DCB in the list.

TSO Macros

Table 30. TSO Macros That CMS Simulates

Macro	Parameters/Options Supported
STAX	
TCLEARQ	
TGET	
TPUT	

Simulated OS/MVS Supervisor Calls

Table 31. Simulated OS/MVS Supervisor Calls

Linkage	Macro	Function
SVC 00	XDAP	Reads or writes direct access volumes
SVC 00	EXCP	Executes a channel program for graphic access method (GAM)
SVC 01	WAIT	Waits for one or more events
SVC 02	POST	Signals event completion
SVC 03	None	Exits from called program
SVC 04	GETMAIN	Acquires user storage
SVC 05	FREEMAIN	Releases user-acquired storage
SVC 06	LINK	Passes control to another program

[14] The DEVTYPE interface will not return valid track or cylinder details that can be used for DASD space calculations. It is intended only to give access to default device characteristics. If detailed real DASD device characteristics are needed, see CP DIAGNOSE X'210' in the *z/VM: CP Programming Services* or the DEVTYPE utility program in the *z/VM: CMS Commands and Utilities Reference*.

Table 31. Simulated OS/MVS Supervisor Calls (continued)

Linkage	Macro	Function
SVC 07	XCTL	Passes control to another program at the same task level and does not return to the calling program
SVC 08	LOAD	Reads a program into storage
SVC 09	DELETE	Deletes a loaded program
SVC 10	GETMAIN/ FREEMAIN	Manipulates user free storage
SVC 11	TIME	Gets the time of day
SVC 13	ABEND	Terminates processing with user specified completion and reason codes
SVC 14	SPIE	Sets up an exit to be given control under user selected program interrupts
SVC 17	RESTORE	NOP
SVC 18	BLDL	Builds a directory list for a partitioned data set
SVC 18	FIND	Locates a member of a partitioned data set
SVC 19	OPEN	Prepares a DCB for I/O processing
SVC 20	CLOSE	Completes and secures I/O processing on a DCB
SVC 21	STOW	Updates partitioned dataset directories
SVC 22	OPEN TYPE=J	Prepares a DCB for I/O processing after an RDJFCB has been issued
SVC 23	CLOSE TYPE=T	Temporarily deactivates a tape or DASD file
SVC 24	DEVTYPE[14]	Obtains device-type physical characteristics
SVC 25	TRKBAL	NOP
SVC 31	FEOV	Forces an EOV condition on a tape or DASD file
SVC 35	WTO/WTOR	Writes a message to the operator's terminal
SVC 40	EXTRACT	NOP
SVC 41	IDENTIFY	Adds an entry name to a loaded program
SVC 42	ATTACH	Passes control to another program at a new task level
SVC 44	CHAP	NOP
SVC 46	TTIMER	Tests or cancels timer
SVC 46	STIMERM	Tests or cancels multiple timer (real time only)
SVC 47	STIMER/STIMERM	Sets the timer interval and the timer exit routine
SVC 48	DEQ	NOP
SVC 51	SNAP	Dumps specified areas of storage
SVC 56	ENQ	NOP
SVC 57	FREEDBUF	Releases simulated BDAM buffer
SVC 60	STAE	Sets up an abend exit routine in a 370 virtual machine

Table 31. Simulated OS/MVS Supervisor Calls (continued)

Linkage	Macro	Function
SVC 60	ESTAE	Sets up an abend exit routine in a 370, XA and XC virtual machine
SVC 62	DETACH	NOP
SVC 63	CHKPT	NOP
SVC 64	RDJFCB	Obtains information from FILEDEF command about an OS/MVS data set
SVC 68	SYNADAF	Provides SYNAD analysis function
SVC 68	SYNADRLS	Releases SYNADAF message and save areas
SVC 69	BSP	Backs up a record on a tape or disk
SVC 93	TGET/TPUT	Reads or writes a terminal line
SVC 94	TCLEARQ	Clears terminal input queue
SVC 96	STAX	Sets or cancels user exit for terminal attention interrupts
SVC 109	ESPIE	Sets up handlers for program interrupts in XA and XC modes
SVC 112	PGRLSE	NOP
SVC 112	PGOUT	NOP
SVC 113	PGLOAD	NOP
SVC 122	LINK, XCTL, LOAD	Performs a link, transfer of control, or load in XA and XC virtual machines
SVC 138	PGSER	NOP

OS/MVS Macros for Assembly Only

In addition to the OS/MVS macros that CMS simulates, CMS includes many nonsimulated OS/MVS macros. The macros are contained in OSMACRO1 MACLIB.

These macros, which are listed below, are for assembly purposes only. Because CMS does **not** simulate these macros you should not use them in programs you intend to run on CMS.

For more information on these macros, see *OS/390 MVS Assembly Service* or *DFSMS Access Service Macros*.

```
ATSET          LOCASCB        SCHEDULE
AXEXT          LXFRE          SDUMP
AXFRE          LXRES          SEGLD
AXRES          MGCR           SEGWT
AXSET          MODESET        SETFRR
BLSABDPL       NIL            SETL
BLSQMDEF       NUCLKUP        SETLOCK
BLSRESSY       OIL            SETPRT
BUILD          PCLINK         SPOST
CALLDISP       PDAB           SRBSTAT
CALLRTM        PDABD          SRBTIMER
CHANGKEY       PGANY          STATUS
CIRB           PGFIX          SUSPEND
CPOOL          PGFIXA         SVCUPDTE
CPUTIMER       PGFREEA        SYNCH
CVT            PROTPSA        SYSEVENT
DATOFF         PRTOV          TCTL
DOM            PTRACE         TESTAUTH
DSGNL          PURGEDQ        TRUNC
DYNALLOC       QEDIT          VRADATA
ECVT           RACDEF         VSMLIST
ESETL          RACHECK        VSMLOC
ETCON          RACINIT        VSMREGN
```

```
ETCRE           RACLIST          WTL
ETDES           RACROUTE         XLATE
ETDIS           RACSTAT
EVENTS          RACXTRT
FESTAE          RELEX
FRACHECK        RESERVE
GQSCAN          RESUME
IOSINFO         RISGNL
IOSLOOK         RPSGNL
```

OS/MVS Resource Management

By default, CMS releases GETMAIN storage and user exit routines at SVC 202/CMSCALL and SVC 42/ATTACH termination. This method is consistent with MVS and provides a programming environment where clearly defined boundaries exist between programs at different SVC levels. (For a detailed discussion of SVC-levels, end-of-command, and program boundaries, see Chapter 2, "CMS Operating Characteristics," on page 15.)

Note: In releases of CMS previous to CMS 5.5, CMS released GETMAIN storage and user exit routines at end-of-command (end-of-command is when CMS displays the ready message (Ready;)).

Cleaning Up GETMAIN Storage

The SET STORECLR command specifies whether CMS cleans up GETMAIN storage and user exit routines at end-of-command or at SVC 202/CMSCALL and SVC 42/ATTACH termination. This, in turn, affects whether boundaries exist between programs at different SVC levels, and it affects how certain functions (the STRINIT macro and EXECOS) work.

The default value SET STORECLR ENDSVC indicates that CMS releases GETMAIN free storage and user exit routines when the current SVC level ends; STRINIT and EXECOS are treated as no-ops.

If you use SET STORECLR ENDCMD, CMS releases GETMAIN free storage and user exit routines at end-of-command; STRINIT and EXECOS are handled as in previous releases.

If an entire system must retain GETMAIN storage, place SET STORECLR ENDCMD command in the SYSTEM profile. If a particular virtual machine must retain GETMAIN storage, place the SET STORECLR ENDCMD command in the user's PROFILE EXEC. For a particular application, you can switch SET STORECLR from ENDCMD to ENDSVC and back using the CMSCALL macro for assembler programs, or using a front-end exec. Note, however, that when you use the SET STORECLR command to change the current setting, a STRINIT is performed on existing OS/MVS GETMAIN storage.

Use the QUERY STORECLR command to determine the current setting of GETMAIN storage cleanup.

The following table shows how other CMS components operate in CMS based on the STORECLR setting.

Table 32. SET STORECLR Command Setting

Component	ENDCMD	ENDSVC
STRINIT	same	No-op
EXECOS	same	No-op
EXEC	same	Resources released at SVC 202/CMSCALL termination
EXEC2/REXX	same	Resources released at SVC 202/CMSCALL termination
PROP	same	Resources released at SVC 202/CMSCALL termination

CMS Storage Management

CMS supports all GETMAIN and FREEMAIN options except for the RELATED keyword; CMS ignores RELATED. The options CMS supports include:

- The LIST options (LC, LU, L)
- Subpool support (SP)
- RC, RU, VRC, VRU - these options enable CMS to use the CMSSTOR LOC parameter to obtain and release storage above the 16MB line.

CMS translates GETMAIN and FREEMAIN requests into an appropriate CMS storage request. This means that CMS supports only one storage management system based on the CMSSTOR and SUBPOOL macros. Programs cannot use NUCON addresses for GETMAIN storage.

You can use the SET GETMAIN command to control the amount of storage obtained on a variable GETMAIN request. When you issue a variable GETMAIN request and there is not enough contiguous free storage to satisfy the entire request, a predetermined percentage of the largest free storage area is allocated by default. The size of the virtual machine determines the percentage as follows:

Table 33. SET GETMAIN Storage Setting

Virtual Machine Size	Percentage of the Largest Free Storage Area Allocated
Less than 4MB	67
Between 5MB and 6MB	80
Greater than 6MB	95

All CMS storage requests are satisfied from the highest available free storage address. If the above algorithm is used followed by smaller GETMAIN requests, those requests will be satisfied from the remaining available storage and could remain in storage long after the variable storage from GETMAIN has been freed. This presents the possibility of fragmenting storage or not being able to load non-relocatable modules that could be loaded prior to release VM/SP 6 and VM/XA* SP. You can use the SET GETMAIN command to change the percentage of GETMAIN storage used to satisfy variable GETMAIN requests, allowing additional control over the GETMAIN process.

CMS Simulation of OS/MVS Subpools

CMS simulation of OS/MVS subpools is very similar to the way subpools work in MVS/XA. One exception is that only subpools 0-127 are valid in CMS; indicating any other subpool results in an abend.

Use the SP parameter on GETMAIN or FREEMAIN to request that CMS create or release a subpool. Make sure that the name you use for the subpool adheres to the CMS subpool naming conventions. (See the *z/VM: CMS Macros and Functions Reference* for a description of these conventions.)

How CMS Handles GETMAIN Storage

By default, CMS releases GETMAIN storage at SVC 202/CMSCALL or SVC 42/ATTACH termination. This removes the need to issue the STRINIT macro. Therefore, when SET STORECLR ENDSVC is specified or defaulted to, CMS treats the STRINIT macro as a no-op.

Note: Prior to CMS 5.5, CMS released GETMAIN storage at end-of-command.

For example,

1. Assume that Program A issues an SVC 202 to call Program B
2. Program B issues a GETMAIN to obtain some free storage
3. When Program B returns to Program A, CMS releases the free storage that Program B obtained.

If you want CMS to release GETMAIN storage at end-of-command, use the SET STORECLR ENDCMD. For the previous example, let's assume that SET STORECLR ENDCMD was issued before Program A was invoked. For this case, when Program B returns to Program A, CMS would not release the free storage

Program B had obtained. Program A would have to issue a STRINIT to explicitly release the free storage that Program B had obtained.

Program Management

1. MVS/XA Linkage Editor at DFP level 2.3.0 provides AMODE and RMODE support.
2. CMS treats the RMODE and AMODE attributes in the same manner as MVS/XA. MVS services that run only in 24-bit addressing mode in MVS/XA run only in 24-bit addressing mode on XA and XC virtual machines. MVS services that run only in 31-bit addressing mode in MVS/XA run only in 31-bit addressing mode on XA and XC virtual machines. MVS services that run in either addressing mode in MVS/XA run in either mode on XA and XC virtual machines.
3. CMS supports 31-bit addressing for the LOAD, LINK, ATTACH, XCTL, and DELETE macros. After completion of a LOAD, bit 0 of register 0 indicates the program's addressing mode. When a program has been called with LINK/ATTACH/XCTL, bit 0 of register 14 indicates the addressing mode of the calling program.

Program Boundaries in OS/MVS Simulation

CMS simulates the MVS task control block and the MVS task block boundaries. CMS makes available and releases OS/MVS exits and programs in a manner similar to MVS. Abend exits are available to any active program within the task. They are cleaned up when the program that issues them terminates.

CMS defines a task at the first SVC level associated with the command or when an OS/MVS ATTACH macro is issued. This is not meant as multitasking; rather, it determines:

- Exit availability for percolation (percolation is the transfer of control between exits).
- When CMS cleans up called programs and exits.

CMSCALL and ATTACH are treated as MVS task block boundaries. Thus, exits such as ESPIE exits and ESTAE exits are available to programs called with CMSCALL as long as the program that issued the exit is still active. Except for STAX exits, exits are cleaned up when the program that issued them ends. STAX exits are available after the program that issues the STAX ends; they survive until task clean up is done by CMS. CMS performs task clean up when the SVC level designated as a task ends. In MVS, STAX exits are cleaned up only at task clean up.

OS/MVS Exit Availability and Clean-Up Behavior

Assume that (as shown in the following example) Program A starts executing, issues some macros, and then calls Program B. Program B issues some macros and then uses the ATTACH macro with ESTAI to call Program C. Program C issues some macros and then does an SVC 202 to Program D. Figure 53 on page 341 shows the exit environment in effect at the point that Program D starts executing. The dotted lines (<---->) represent the task boundaries (similar to the MVS task control block boundary) that separate the programs. Programs cannot access the exit routines across a task boundary. For example, the exits available to Program D are:

ESPIE C1
ESTAE D1
ESTAI C1
STAX D1

When Program D ends, the ESTAE exit it established is cleaned up. The STAX exit it established remains available until Program C completes, since the ATTACH macro defined a task boundary to CMS. For more specific information about the individual exits and their clean up (request block/program level or task) see the *MVS/XA Supervisor Services and Macro Instructions* book.

```
   MACROS ISSUED        EXIT ENVIRONMENT
<------------------------------------------------------------>

 PROGRAM A             POINTER->ESTAE-A2-->ESTAE-A1
   ESTAE-A1
   ESTAE-A2
   CMSCALL PROGRAM B

 PROGRAM B
   STIMERM-B1          POINTER->STIMERM-B1->STIMERM-B3->STIMERM-B2
   STIMERM-B2          POINTER->ESPIE-B1
   STIMERM-B3          POINTER->ESTAE-B1
   ESPIE-B1
   ESTAE-B1
   ATTACH PROGRAM C, ESTAI-C

<------------------------------------------------------------>

 PROGRAM C

  ESPIE-C1             POINTER->ESPIE-C1
  SVC 202 to PROGRAM D

 PROGRAM D

  ESTAE-D1             POINTER->ESTAE-D1->ESTAI-C
  STAX-D1              POINTER->STAX-D1

<------------------------------------------------------------>
```

Figure 53. OS/MVS Exit Availability and Clean-Up Behavior

Note: The exits defined by Program A and Program B (for example, ESTAE-A1 and ESPIE-B1) are not available across the boundary established by the ATTACH. The ESTAI parameter can be used on ATTACH to intercept abnormal termination. Any programs called after the ATTACH can also use the ESTAI-C1 exit program.

When several exits are defined for a program, the order of availability is the same as MVS. For SPIE, ESPIE, STAE, ESTAE, ESTAI, and STAX, a LIFO order is followed. The transfer of control between exits is called **percolation**. Percolation happens with STAE/ESTAE and STAI/ESTAI exits. During abend processing, CMS checks SPIE/ESPIE exits before STAE/ESTAE or STAI/ESTAI exits. If control is given to a SPIE/ESPIE exit, no further exit processing is done in CMS. For STIMERM and STIMER, the order of availability is determined by the time interval requested as in MVS. The pointers (->) in Figure 53 on page 341 illustrate the order of availability of exits.

Note:

1. ATTACH processing in CMS differs from the same process in an OS environment. When the ATTACH macro is issued, Program B halts execution until Program C completes and returns through an OS Simulated Exit (SVC 3) or a branch instruction. Program B has control returned to it at the instruction following the ATTACH of Program C, with a return code in Register 15 and in the ECB (if specified), even if there is a program check or the ABEND macro is issued during the execution of Program C or Program D. System return codes 15A and 155 will be generated when CMS is not able to locate the requested text deck or module respectively. (For more information on OS loading modules, refer to the CMS COMPSWT macro in the *z/VM: CMS Application Development Guide*.) When CMS is able to locate the program in the ATTACH macro (in Program C) and that program exits, CMS updates Register 15 of Program B and the ECB return code (if ECB was specified on the ATTACH macro) with the last value of Register 15 in Program C.
2. SET STORECLR ENDCMD should be used with caution as it effectively limits the user to one globally defined task for the life of the active command. If EXECOS is issued at any point within the command, all OS/MVS resources (such as exits, maclibs, and loadlibs) are freed without any regard for task or request block boundaries. If EXECOS is issued in the CMS default environment (SET STORECLR ENDSVC), it resets DOS and VSAM pointers only if they are active. It does not affect OS/MVS resources. CMS releases OS/MVS resources during program (request block) or task level clean up.

Abnormal Termination

1. To exploit 31-bit addressing for program interrupt handling or abend handling, CMS supports the ESPIE, ESTAE, and SETRP macros, and the ESTAI parameter of the ATTACH macro.

   For ESPIE, an EPIE work area is provided. For ESTAE, an SDWA work area is provided.
2. CMS supports ABEND macro reason codes. (These reason codes are passed to the exit routine using the SDWA.)
3. CMS does not fully support task control block/request block (TCB/RB) environments for ESPIES as in OS/MVS. For example, CMS allows a program to override/delete a specific SPIE/ESPIE environment established under a previous TCB.

Timer Support

CMS simulates the MVS/XA STIMERM macro to provide multiple timer interval support. STIMERM is supported in all virtual machine modes.

STIMER and STIMERM allow you to set a timer interrupt request for a specified time interval or for an interval that expires at a specified time of day. Up to 16 such requests may be in effect at a time with STIMERM (total of 17 per task with STIMER).

Using CMS Libraries

CMS provides three types of libraries to aid in OS/MVS program development:

- Macro libraries contain macro definitions, copy files, or both
- Text, or program libraries, contain object programs (compiler output)
- LOADLIB libraries contain link edit files (load modules).

These CMS libraries are functionally like OS/MVS partitioned data sets (PDS); each has a directory and members. These libraries are stored in a data format unlike either a real MVS PDS or a normal CMS file. Because they are not like other CMS files, you create, update, and use them differently than you do other CMS files. Although these library files are similar in function to OS/MVS partitioned data sets, OS/MVS macros should not be used to update them. The *z/VM: CMS Application Development Guide* discusses macro libraries and text libraries. The following sections discuss LOADLIB libraries.

OS/MVS Module Libraries and CMS LOADLIBS

The OS/MVS relocating loader allows the user to load a member of a CMS LOADLIB or an OS/MVS module library on an OS/MVS formatted disk. The OS/MVS LINK, LOAD, ATTACH, and XCTL macros are supported. In addition, the OSRUN command (which generates a LINK SVC) loads and executes members directly from the console.

For the LINK, LOAD, ATTACH, and XCTL macros, the libraries specified in the LOADLIB global list are searched. If the requested member is not found, CMS looks for a TEXT file by that name. Then, if still not found, the TXTLIBs specified in the TXTLIB global list are searched for the member name.

For the OSRUN command, the libraries specified in the LOADLIB global list are searched. If the member is not found and the user has a $SYSLIB LOADLIB file, it is searched for the member name. (TEXT files and TXTLIBs are not considered by OSRUN.)

You can use the PARM operand to pass OS parameters to the module in the CMS LOADLIB or OS/MVS module library. If the parameters contain blanks or special characters, they must be enclosed in quotation marks. To include quotation marks in the parameters, use double quotation marks. The parameters are passed in OS format: register 1 points to a fullword containing the address of a character string headed by a halfword field containing the length of the character string. The parameters are restricted to a maximum length of 100 characters.

Note: You may not pass parameters (PARM=) to the module if you issue the OSRUN command from a CMS EXEC file. The OSRUN command can be issued from a REXX or an EXEC 2 file with no restrictions.

Executing OS/MVS Module Libraries

If the module to be executed resides in an OS/MVS module library on an OS/MVS formatted disk, the disk must be accessed and the library must be defined (through the FILEDEF command) to make it known to CMS.

For example, access the OS/MVS disk as a B-disk at the address 250:

```
ACCESS 250 B
```

Suppose SYS1.TESTLIB is an OS/MVS module library on the OS/MVS disk and contains the member TEST1.

Use the FILEDEF command to relate SYS1.TESTLIB to the CMS LOADLIB called OSLIB LOADLIB:

```
FILEDEF $SYSLIB DISK OSLIB LOADLIB B DSN SYS1 TESTLIB
     (DSORG PO RECFM U BLOCK 7294
```

The ddname specified on the FILEDEF command must be $SYSLIB. The file name (OSLIB), specified on the FILEDEF command, can be any name. The file type must be LOADLIB.

After issuing this FILEDEF, you can refer to the OS module library, SYS1.TESTLIB, by using the CMS file identifier OSLIB LOADLIB.

Before you try to execute TEST1, use the GLOBAL command to identify the CMS LOADLIBs to be searched. The library name must correspond to the file name specified in the FILEDEF command. For example,

```
GLOBAL LOADLIB OSLIB
```

Then, the OSRUN command searches OSLIB LOADLIB for the member, TEST1, to load and execute. For example,

```
OSRUN TEST1
```

Creating and Executing CMS LOADLIBs

If the program to be executed resides on a CMS disk, use the LKED command. The LKED command creates a CMS LOADLIB from a CMS TEXT file. For example:

```
LKED TESTFILE
```

takes the CMS TEXT file, TESTFILE TEXT, and creates the CMS LOADLIB, TESTFILE LOADLIB. The CMS LOADLIB created by the LKED command is an OS simulated partitioned data set (PDS) named TESTFILE LOADLIB and contains one member named TESTFILE.

For more information on input to the LKED command refer to "The LKED Command" on page 344.

Before executing TESTFILE, use the GLOBAL command to identify the LOADLIB to be searched:

```
GLOBAL LOADLIB TESTFILE
```

Then the OSRUN command loads, relocates, and executes the TESTFILE member of TESTFILE LOADLIB:

```
OSRUN TESTFILE
```

Maintaining CMS LOADLIBs

The LOADLIB command provides the utility necessary to maintain the CMS LOADLIBs. The following parameters are provided:

COPY
: Copy members from one LOADLIB to another Merge complete LOADLIBs Copy with SELECT or EXCLUDE

COMPRESS
Compress a CMS LOADLIB

LIST
LIST members of a CMS LOADLIB

For more detailed information on the LKED, GLOBAL, OSRUN, and LOADLIB commands, see the *z/VM: CMS Commands and Utilities Reference*.

Concatenating Files

To define more than one library with the same ddname, use the CONCAT option of the FILEDEF command. You can concatenate the LOADLIB files on OS/MVS disks with each other or with CMS LOADLIB files. Any library to be searched must be specified in the GLOBAL LOADLIB statement. The data set with the largest block size should be specified first (both in the FILEDEF and in the GLOBAL list). CMS files do not require a file definition. The GLOBAL list determines the order in which the libraries are searched.

For GLOBAL libraries, the filemode on a related concatenated FILEDEF is honored. If the file cannot be found on the specified cms disk, an error message is issued during open processing for the ddname. If a filemode of '*' is used on FILEDEFs relating to GLOBAL libraries, the established search order finds the first occurrence of the file and uses it. However, the use of filemode '*' can increase search time and degrade overall performance if there are many disks in the search order.

For example, search two OS/MVS files and a CMS LOADLIB for the member, THETA, using the following commands:

```
ACCESS 250 B (if 250 is the address of the OS/MVS disk)
FILEDEF $SYSLIB DISK OSLIB LOADLIB DSN SYS1 LIB1
     (DSORG PO RECFM U BLOCK 7294)
FILEDEF $SYSLIB DISK MYLIB LOADLIB B DSN SYS1 LIB2 (CONCAT)
GLOBAL LOADLIB OSLIB MYLIB DMSGPI
OSRUN THETA
```

If any library in a global list (or concatenation) resides on an OS/MVS format disk, then the first library in the concatenation for those global libraries must reside on an OS/MVS format disk. For example, if both CMS and MVS LOADLIBs are to be used in the same concatenation, one of the MVS LOADLIBs must be the first library in the concatenation. The remaining CMS and MVS LOADLIBs can follow in any order desired.

For example, the following sequence of commands may be used:

```
FILEDEF $SYSLIB DISK OSLIB1 LOADLIB * DSN SYS1 LIB1
     (DSORG PO BLOCK 2320)
FILEDEF $SYSLIB DISK OSLIB2 LOADLIB * DSN SYS1 LIB2
     (DSORG PO CONCAT BLOCK 2320)
FILEDEF $SYSLIB DISK TESTLIB1 LOADLIB *
     (DSORG PO CONCAT BLOCK 2320)
GLOBAL LOADLIB OSLIB1 OSLIB2 TESTLIB1
```

The first FILEDEF command for $SYSLIB should describe the first library file name in the GLOBAL list. Its attribute is used when the libraries are searched. It is advisable not to code the CONCAT option on the first FILEDEF command because it clears all previous FILEDEFs for that ddname.

For a file in a GLOBAL list, FILEDEF attempts to open the file on the specified disk. If the file cannot be found on the specified disk, an OPEN error message is issued.

Using filemode * degrades performance of concatenated files.

The LKED Command

The LKED command uses the MVS/XA Linkage Editor for the actual link of the TEXT file to the LOADLIB as an executable module. To link edit CMS files, you can issue the FILEDEF command to identify input to the MVS/XA Linkage Editor. Primary LKED input is a data set known to the linkage editor as SYSLIN, which can be described in the *fname* operand of the LKED command. The file type of the input file named in the command line must be TEXT. Optionally, you can override the *fname* operand by issuing a FILEDEF that defines SYSLIN as the ddname of an alternate primary input source. If your alternate input is a CMS file, the choice of file type is unrestricted. The contents of the SYSLIN dataset may be:

1. Object text such as assembler or compiler output
2. Linkage editor control statements
3. A combination of object text and control statements.

Linkage editor control statements can be inserted before, between, and after object modules and other control statements. Editing procedures can be used to construct files to meet your requirements. Linkage editor INCLUDE statements may be used to designate explicitly the following files or file members as secondary linkage editor input:

1. CMS TEXT files
2. Members of CMS TXTLIB files
3. Members of CMS LOADLIB files
4. Members of OS/MVS object libraries
5. Members of OS/MVS load libraries.

A FILEDEF must be issued before the LKED command to define a unique ddname for each file to be included as secondary linkage editor input. An INCLUDE statement in the SYSLIN dataset must specify the ddname assigned to the file by your FILEDEF. For library files, the statement must also specify all members of the library that are to be included as input. The following example shows all FILEDEF commands and INCLUDE statements used to identify input files.

CMS commands:

```
FILEDEF LIBDEF DISK MYLIB TXTLIB B
FILEDEF TXTDEF DISK MYFILE TEXT C
LKED INPUT (LIBE TESTLIB
```

SYSLIN input:

```
INCLUDE LIBDEF(CSECT1,CSECT2)
INCLUDE TXTDEF
NAME TESTPROG
```

The LKED command links CSECT1 and CSECT2 from the MYLIB TXTLIB with the MYFILE TEXT file, builds TESTLIB LOADLIB, and puts the executable module, TESTPROG, in TESTLIB LOADLIB. The INPUT TEXT file contains the SYSLIN input.

INCLUDE statements must begin in column 2. The applicable statement formats are described in the *MVS/XA Linkage Editor and Loader User's Guide*.

Use the following CMS commands to execute TESTPROG:

```
GLOBAL LOADLIB TESTLIB
OSRUN TESTPROG
```

When SYSLIN input to the LKED command is an assembled object file in fixed-block format residing on an OS/MVS disk, the RECFM FBS option of the FILEDEF command must be specified.

Example of identifying an OS/MVS object library and CMS TXTLIB

The following FILEDEF commands and SYSLIN input identify a member of an OS/MVS object library and a CMS TXTLIB.

CMS commands:

```
FILEDEF OSOBJ DISK OBJECT FILE Q DSN SYS1 FEOBJ (RECFM FBS
 LRECL 80 BLOCK 3120
FILEDEF TXTDEF DISK NEWLIB TXTLIB B
```

SYSLIN input:

```
INCLUDE OSOBJ(MEMBER1)
INCLUDE TXTDEF(CSECT1)
```

Automatic library search is available for either CMS or OS/MVS type library members if the FILEDEF for the dataset to be searched specifies SYSLIB as the ddname. Additional libraries can be selected for automatic search by placing linkage editor LIBRARY statements in your SYSLIN input file. Each library statement must contain the associated ddname and a list of members within the library to be included in the search. A FILEDEF must be issued before the LKED command to assign a unique ddname to each dataset to be searched. The CONCAT option of the FILEDEF command is valid for LKED input datasets only when the ddname is SYSLIB and the input file type is TXTLIB. To expand the automatic SYSLIB search, the user may combine the members of several CMS libraries into a single composite library. The automatic search facility applies to CMS TXTLIBs and LOADLIBs and to OS object libraries and LOAD libraries.

Example of automatic library search and concatenating CMS TXTLIBs

The following example shows FILEDEF commands and SYSLIN input for an automatic library search and concatenating CMS TXTLIBs as input to the LKED command.

CMS commands:

```
GLOBAL TXTLIB SEARCH1 SEARCH2 SEARCH3
FILEDEF SYSLIB DISK SEARCH1 TXTLIB A (CONCAT
FILEDEF LIBDEFA DISK SEARCH4 LOADLIB (RECFM U
FILEDEF LIBDEFB DISK OSTEXT LIBRARY D DSN OBJMODS
```

SYSLIN input:

```
LIBRARY LIBDEFA(MEMBER1,MEMBER2)
LIBRARY LIBDEFB(MEMBER3,MEMBER4)
```

LIBRARY statements must begin in column 2. The GLOBAL command is only needed to identify concatenated TXTLIBs as input to the linkage editor. It is not needed for those libraries specified in the linkage editor LIBRARY statements. You cannot use SYSLIB for TXTLIB and LOADLIB in the same LKED session. For LOADLIB input to the linkage editor, the RECFM U option of the FILEDEF command must be specified.

As shown in the example, to concatenate TXTLIBs as input on the LKED command, the ddname on the FILEDEF command must be SYSLIB. The file name, SEARCH1, specified on the FILEDEF command, can be any valid TXTLIB file name. (This file name does not have to correspond to any file names listed on the GLOBAL command.) The file type must be TXTLIB, and the file must be a fixed-formatted file.

If you concatenate TXTLIBs, all the TXTLIBs listed on the GLOBAL command are searched and the TXTLIB specified on the FILEDEF command is ignored. The GLOBAL command determines the order in which the libraries are searched. If you do not concatenate TXTLIBs, you do not need to issue a GLOBAL command. Only the TXTLIB specified on the FILEDEF command is searched.

The default FILEDEF commands issued by the LKED command for the ddnames presented to the Linkage Editor are as follows:

```
FILEDEF SYSLIN DISK FNAME TEXT * (RECFM F BLOCK 80 NOCHANGE
FILEDEF SYSLMOD DISK fname LOADLIB A1 (RECFM U BLOCK 260 NOCHANGE
-or-
FILEDEF SYSLMOD DISK libname LOADLIB A1 (RECFM U BLOCK 260 NOCHANGE
FILEDEF SYSUT1 DISK fname SYSUT1 *
FILEDEF SYSPRINT DISK fname LKEDIT A1
-or-
FILEDEF SYSPRINT PRINTER
-or-
FILEDEF SYSPRINT DUMMY
```

At the completion of the LKED command, all FILEDEFs that do not have the PERM option are erased.

Example of linking a program that requires more than one library

In the following example, your assembler program, TEST TEXT, calls a routine, SUB0000, from a user library, USERLIB TXTLIB. TEST TEXT also calls routines from the MYLIB library. Then, the LKED command places an executable module called TEST0000 in the TESTLIB LOADLIB.

You can define and create the appropriate libraries using one of the following two methods:

Method 1

First, create the following SYSLIN input file called INPUT TEXT:

```
 INCLUDE TEXTDEF
 LIBRARY LIBDEF(SUB0000)
```

Next, enter the following CMS commands:

```
FILEDEF TXTDEF DISK TEST TEXT A
FILEDEF SYSLIB DISK MYLIB TXTLIB *
FILEDEF LIBDEF DISK USERLIB TXTLIB *
LKED INPUT (LIBE TESTLIB NAME TEST0000
```

To execute TEST0000, enter the commands:

```
GLOBAL LOADLIB TESTLIB
OSRUN TEST0000
```

Note: Note that by using the library statement, instead of the INCLUDE statement, the indicated member(s) are only added to the LOADLIB as they are required by LKED to resolve external references found in the program. The INCLUDE and LIBRARY statements must begin in column 2.

Method 2

Enter the following CMS commands:

```
GLOBAL TXTLIB USERLIB MYLIB
FILEDEF SYSLIB DISK USERLIB TXTLIB * (CONCAT
LKED TEST (LIBE TESTLIB NAME TEST0000
```

To execute TEST0000, enter the commands:

```
GLOBAL LOADLIB TESTLIB
OSRUN TEST0000
```

OS/MVS and CMS Terminology

CMS uses many OS/MVS terms, but there are several OS/MVS functions that CMS performs somewhat differently. See Table 34 on page 347 to help you become familiar with some of the equivalents (where they do exist) for OS/MVS terms and functions. It lists some commonly used OS/MVS terms and discusses how CMS handles the functions they imply.

Table 34. OS/MVS Terms and CMS Equivalents

OS Term/Function	CMS Equivalent
cataloged procedure	EXEC files can execute command sequences similar to cataloged procedures, and provide for conditional execution based on return codes from previous steps.
data set	Data sets are called files in CMS. CMS can simulate certain OS/MVS data sets and can read real OS/MVS data sets only if they are sequential or partitioned. CMS can never write to real OS/MVS data sets. CMS reads and writes DOS VSAM data sets.
data definition (DD) card	The FILEDEF command lets you perform the functions of the JCL DD statement to specify device types and output file dispositions.
data set control block (DSCB)	Information about a CMS disk file is contained in a file status table (FST).
EXEC card	To execute a program in CMS you specify only the name of the program if it is an exec, MODULE file, or CMS command. To execute TEXT files, use the LOAD and START commands.

Table 34. OS/MVS Terms and CMS Equivalents (continued)

OS Term/Function	**CMS Equivalent**
job control language (JCL)	CMS and user-written commands perform the functions of JCL.
job step	Command or Exec
link-editing	The CMS LKED command creates LOADLIB libraries from CMS TEXT files or OS/MVS object modules. The CMS LOAD command loads TEXT files into virtual storage, and resolves external references; the GENMOD command creates MODULE files.
load module	Load modules are members of CMS LOADLIB libraries. LOADLIB members are loaded, relocated, and executed by the OSRUN command, and LOADLIB members are loaded and relocated by the NUCXLOAD command. Also, LOADLIB members are referenced by the LINK, LOAD, ATTACH and XCTL macros.
object module	Assembler and high-level language compiler output is placed in CMS files with a file type of TEXT.
OS/MVS	Refers to the OS and MVS operating systems.
STEPCAT, JOBCAT	VSAM catalogs can be assigned for jobs or job steps in CMS by using the special ddnames IJSYSCT and IJSYSUC when identifying catalogs.
partitioned data set	CMS LOADLIBs, MACLIBs, DOSLIBs, and TXTLIBs are CMS files which resemble partitioned data sets.
STEPLIB, JOBLIB	The GLOBAL command establishes macro, text, and LOADLIB libraries; you can indirectly provide job libraries by accessing and releasing CMS disks that contain the files and programs you need.
task	SVC or program level
utility program	Functions similar to those performed by the OS/MVS utility programs are provided by CMS commands.
volume table of contents (VTOC)	The list of files on a CMS disk is contained in a file directory.

Chapter 23. Using OS/MVS Simulated Data Sets in CMS

This chapter describes:

- OS/MVS record formats that CMS can simulate
- Identifying I/O files and devices to OS simulation
- Using OS/MVS simulated data sets and OS/MVS data sets in CMS
- Accessing data through OS simulation
- Using the OS/MVS simulated buffering techniques
- Opening, reading, writing and closing data sets
- OS/MVS exit routines simulated in CMS
- End-of-volume processing.

Overview of OS/MVS Simulated Data Sets

A data set is a collection of logically related data records that are stored on a volume. Data sets are called files in CMS. In this chapter, data set has the same meaning as a file in CMS.

OS simulation lets you define your data sets three ways:

- OS/MVS simulated data sets - resides on CMS format DASD and has a file mode number of 4
- OS/MVS data sets - resides on OS/MVS format DASD
- CMS format files - resides on CMS format DASD.

You can read or write OS/MVS simulated data sets using OS simulation, but, you can only read OS/MVS data sets.

To read or write data sets using OS simulation, you must describe the data to be processed, identify the access method, and identify the buffering technique to be used.

- To find out more about describing data, see "Data Set Organization" on page 349, "Record Formats in OS Simulation" on page 350, "Identifying I/O Files and Devices to OS Simulation" on page 357, "Using OS/MVS Simulated Data Sets in CMS" on page 363, and "Using OS/MVS Data Sets in CMS" on page 368.
- To find out more about access methods, see "Accessing Data through OS/MVS Simulation" on page 373.
- To find out more about buffering techniques, see "Using the OS/MVS Simulated Buffering Techniques" on page 375.

Data Set Organization

CMS can use OS/MVS data that is organized in three ways:

- *Sequential*: Records are organized in physical rather than logical sequence. Given one record, the location of the next record is determined by the physical position in the data set. You must use the sequential data set organization for all magnetic tape devices, but it is optional on direct access devices.
- *Partitioned*: Independent groups of sequentially organized records, called members, are in direct access storage. Each member has a simple name stored in a directory that is part of the data set.
- *Direct*: Records within the data set, which must be on a direct access volume, may be organized in any manner you choose. You specify addresses by which records are stored and retrieved directly.

For more information on sequential, partitioned, and direct data set organization, see the *MVS/XA Data Administration Guide*.

Record Formats in OS Simulation

Sequential, partitioned, and direct data sets consist of records. A record is the basic unit of information used by a processing program. In OS simulation, records can be in one of three formats:

- Fixed-length records
- Variable-length records
- Variable-length spanned records.

Blocking is the process of grouping records before they are written on a volume. A block consists of one or more logical records written between consecutive interrecord gaps (IRGs). For more information on blocking, see "Blocking for CMS Format Files" on page 365.

You can identify your record format in the data control block using options in the DCB macro or in the FILEDEF command. For example, to specify the file, APPLE, as a variable spanned record, you could use the command:

```
FILEDEF APPLE DISK APPLE FILE A4 (RECFM VS
```

Or, you could use the macro:

```
DCB   DDNAME=APPLE,RECFM=VS,DSORG=PS,MACRF=(GL)
```

Fixed-Length Records

The size of fixed-length (format-F) records is constant for all records in the data set. If the records are blocked, then the number of records within a block is constant for every block in the data set, unless the data set contains a truncated (short) block as the last block of the file. If the data set contains unblocked format-F records, one record constitutes one block. See Figure 54 on page 350 for an illustration.

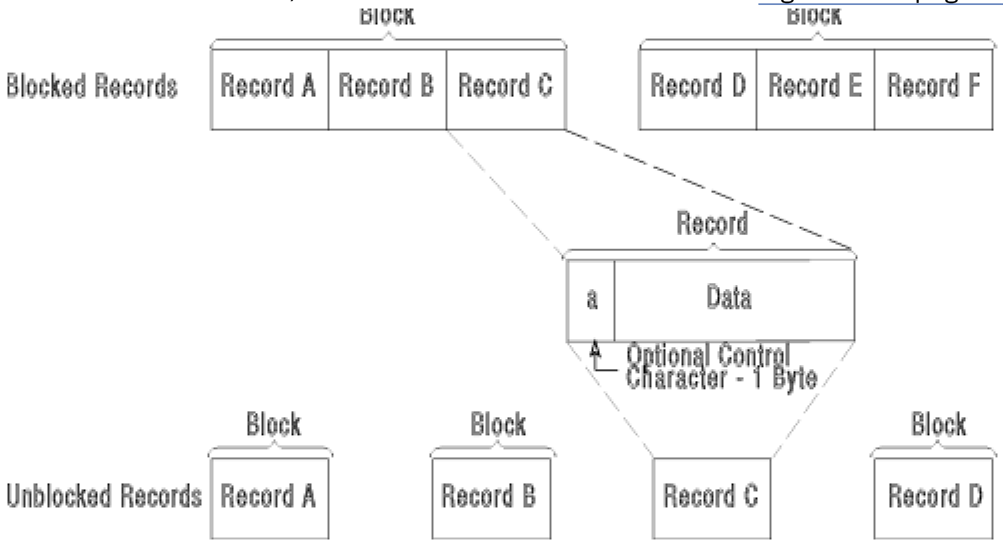


Figure 54. Fixed-Length Records

To indicate that a file consists of fixed-length records, use the F, FB, FS, or FBS options on the FILEDEF command or DCB macro.

```
FILEDEF FRUIT DISK PEAR EXEC A4 (RECFM F
```

indicates that the file, PEAR EXEC, has a fixed-length record format.

The options you can specify with RECFM have the following characteristics:

F
: specifies that the data set contains fixed-length records.

B
specifies that the data set contains blocked records.

S
specifies that the records are written as standard blocks. Standard blocks must conform to the following specifications:

- All records in the data set are format-F records.
- No block except the last block is truncated.
- Every track except the last contains the same number of blocks.
- The data set organization is physical sequential.

Do not code **S** for fixed-length records that were created using a record format other than standard.

Fixed-Length ANSI Records

ANSI records can only be read from and written to ANSI tapes. Fixed-length Format-F records that exist on ANSI tapes are stored in 7-bit ASCII code. Fixed-length ANSI records and blocks are similar to those shown in Figure 54 on page 350 except they may include a block prefix. The block prefix precedes the data and can vary in length from 0 to 99 bytes. See Figure 55 on page 351 for an illustration.

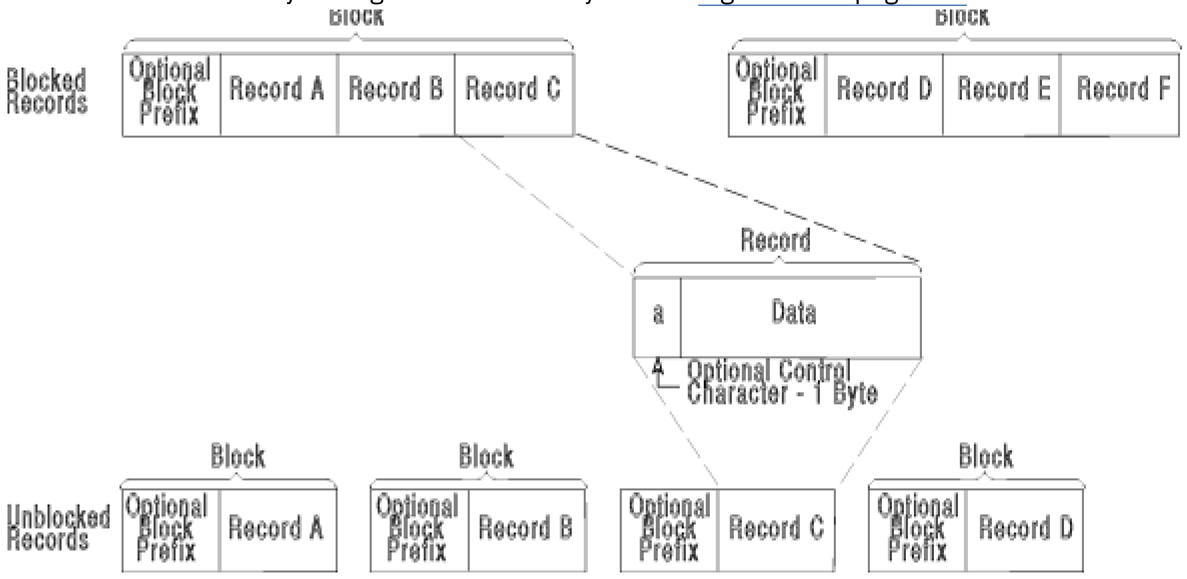


Figure 55. Fixed-Length ANSI Records

You can use the QSAM or BSAM access method to read a fixed-length record. Using QSAM or BSAM to read records with block prefixes requires that you specify the BUFOFF operand in the DCB. The block prefix can be 0 to 99 bytes long. When using QSAM, you cannot access the block prefix on input. When using BSAM, you must account for the block prefix on input and output.

You can use the F or FB option of RECFM to specify fixed-length ANSI records. For example, to be read from an ANSI tape, you could use the following DCB:

```
DCB DSORG=PS,DDNAME=ORANGE,MACRF=(GL),
    LRECL=100,BLKSIZE=120,BUFOFF=20,RECFM=F
```

In this DCB, BUFOFF specifies the length of the block prefix. This length is included in the length specified in the BLKSIZE. RECFM=F indicates that the data contains fixed length records.

When writing to an output file, BUFOFF must be 0 or L. The option L specifies that a block descriptor word of 4 bytes will be written. The only block prefix that OS simulation will write is a BDW.

For more information on the block prefix and ANSI records, see the *MVS/XA Data Administration Guide*.

Variable-Length Records

Format-V provides for variable-length records and variable-length record segments. The first 4 bytes of each record, record segment, or block, make up a descriptor word containing control information. You must allow for the additional 4 bytes in both your input and output buffers.

Block descriptor word

A variable-length block consists of a 4-byte block descriptor word (BDW) followed by one or more logical records or record segments. The first 2 bytes in the BDW specify the block length and the second 2 bytes are reserved. If you do your own blocking, you must supply the BDW. For variable blocks, the minimum buffer length is 9 bytes.

Record descriptor word

A variable-length logical record consists of a 4-byte record descriptor word (RDW) followed by the data. The first 2 bytes contain the length of the logical record. The last 2 bytes must be 0 because these are used for spanned records. When using variable-length records on output, you must provide the RDW; for input, the operating system provides the RDW.

Figure 56 on page 352 shows blocked and unblocked variable length records without spanning.

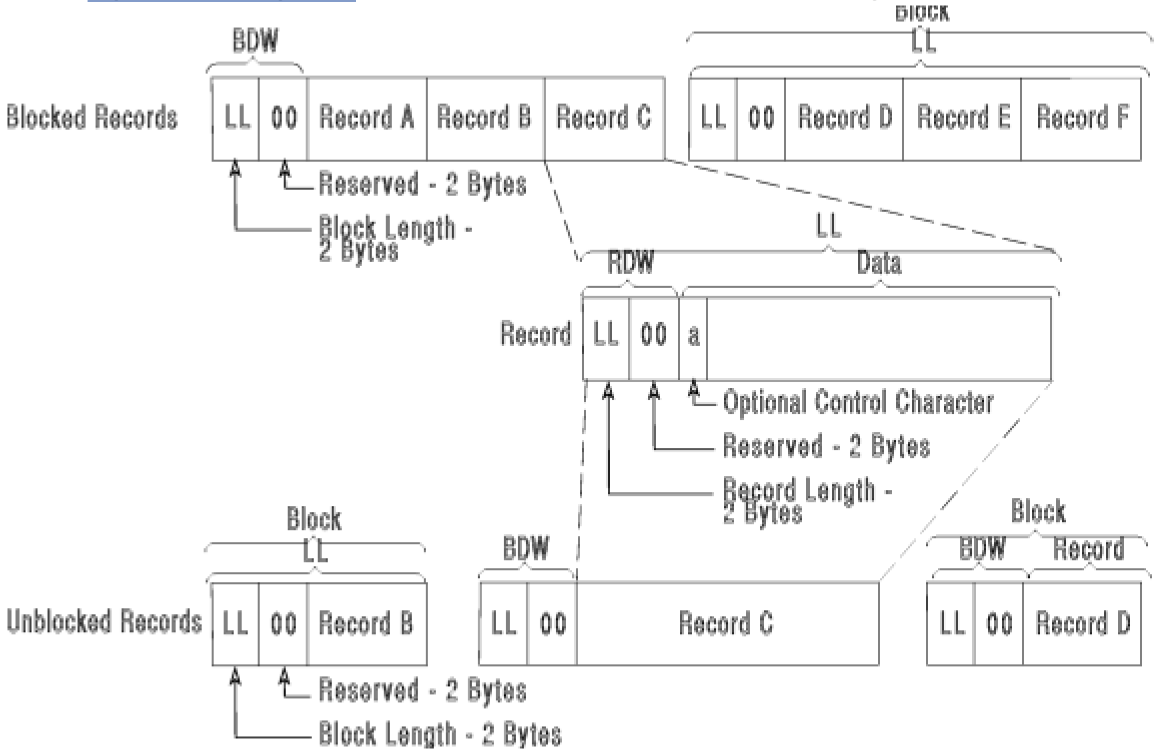


Figure 56. Nonspanned, Format-V Records

See the *MVS/XA Data Administration Guide* for more information on descriptor words.

To specify that a file consists of variable-length records, use the V or VB option on the DCB macro or FILEDEF command. For example,

```
DCB  DDNAME=BANANA,RECFM=VB,DSORG=PS,MACRF=(GL)
```

indicates that the file, BANANA, contains variable blocked records.

The V or VB options have the following characteristics:

V
: specifies that the data set contains variable-length records.

B
specifies the data set contains blocked records.

Variable-Length ANSI Records

Variable-length ANSI records can only be read from and written to ANSI tapes. Variable-length records that exist on ANSI tapes are recorded in 7-bit ASCII code. These records are called format-D records.

Variable-length ANSI records are similar to those shown in Figure 56 on page 352 except they contain a record control word (RCW) instead of an RDW. The RCW is a 4-byte field that describes the record and is internally equivalent to the RDW. (See "Record descriptor word" on page 352 for more information on RDW.) Variable-length ANSI records may also contain a block prefix. See Figure 57 on page 353 for an illustration.

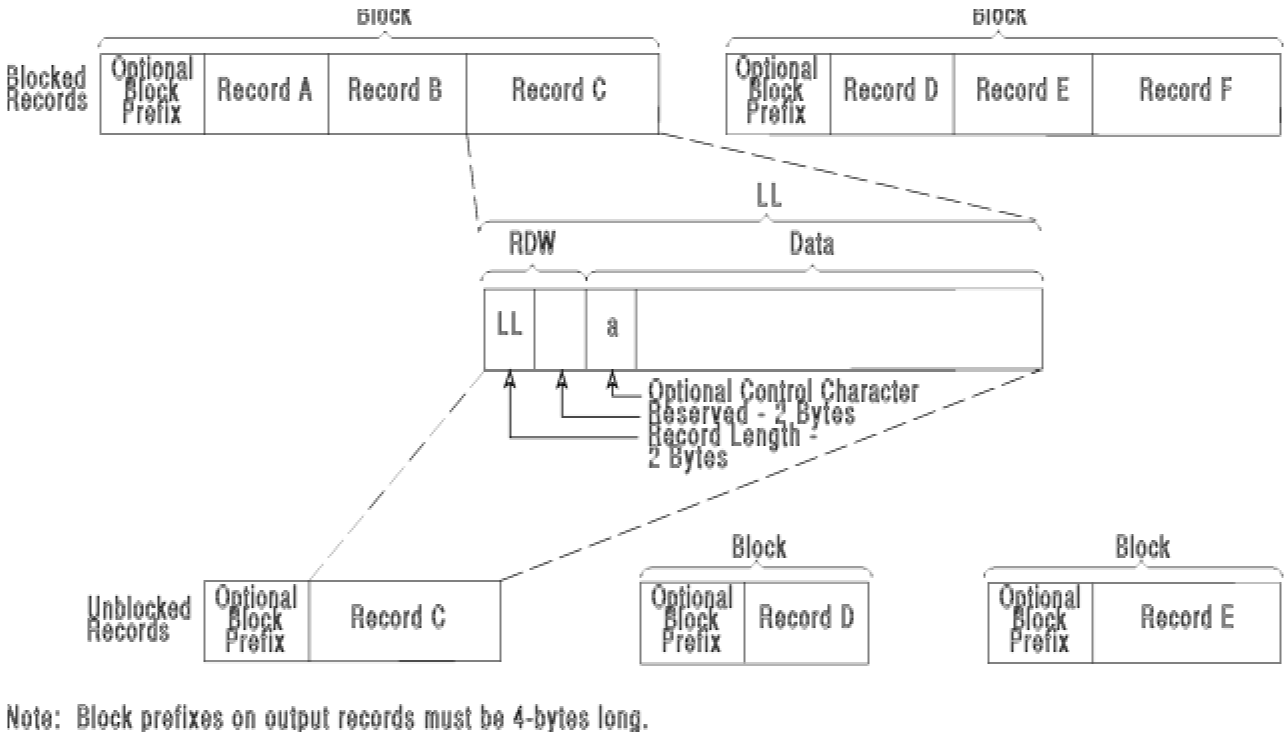


Figure 57. Nonspanned, Format-D Records for ANSI Tapes

To specify that a data set contains variable-length ASCII tape records, use the D option of the RECFM parameter. You can also use the DB option to indicate variable blocked ANSI records. RECFM can either be specified on the FILEDEF command or the DCB macro. Support for variable ANSI records is available through both QSAM and BSAM access methods.

Following is an example of a DCB that specifies variable-length ANSI records.

```
DCB DSORG=PS,DDNAME=ORANGE,MACRF=(GL),
    LRECL=100,BLKSIZE=1020,BUFOFF=20,RECFM=DB
```

The BUFOFF parameter indicates a block prefix will be read. When using QSAM access method for input, the block prefix cannot be returned to the application. When writing to an output file using QSAM or BSAM, BUFOFF must be 0 or L. In CMS, you can only write a block prefix using L. For variable records, using L specifies that you want to write a record descriptor word (RDW).

The length specified for BLKSIZE in the DCB includes the length of the block prefix. For format-D, the minimum value for BLKSIZE or BUFL for variable-length ANSI records BUFL is 18 bytes, the maximum value is 9999 bytes.

For more information about variable-length ANSI records, see the *MVS/XA Data Administration Guide*.

Variable Spanned Records

Variable spanned records are logical records that extend across more than one block. CMS supports spanned records in QSAM and BSAM. With spanned records, a record larger than the block size is split into segments and written in two or more blocks.

- For unblocked records, each block contains only one record or one segment.
- For blocked records, a block can contain a combination of records and segments. However, a block will never contain more than one segment from the same logical record.

Logical record length for spanned records must not exceed 65535.

Since the record length is not dependent on block size, the block size can be set to the one that is best for a given device or processing situation. Block size is not restricted by the maximum record length of a data set.

When unit record devices, such as punches, printers, and readers, are used with spanned records, the system assumes that unblocked records are being processed and the block size must be equivalent to the length of one print line or one card. In this case, records that span blocks are written one segment at a time.

You can specify variable spanned records using the FILEDEF command or DCB macro. For example,

```
FILEDEF PEACHES DISK PEACH FRUIT A4 (RECFM VBS
```

specifies that the file with a *ddname* of PEACHES has variable blocked spanned records. Note that when a CMS DASD device is used, the file containing the variable spanned records must have a file mode number of 4.

Segment descriptor word

Each record segment in a spanned record consists of a segment descriptor word (SDW) followed by the data. The segment descriptor word, similar to the RDW, is a 4 byte field that describes the segment. See "Record descriptor word" on page 352 for information on RDWs.

The first 2 bytes contain the length of the segment, including the 4-byte SDW. The third byte of the SDW contains the segment control code, which specifies the relative position of the segment in the logical record. The segment control code is in the rightmost 2 bits of the byte. (The segment control codes are listed in Table 35 on page 354.) The remaining bits of the third byte and all of the fourth byte are reserved and must be 0.

Table 35. Segment Control Code for SDW

Binary Code	**Relative Position of Segment**
00	Complete logical record
01	First segment of a multisegment record
10	Last segment of a multisegment record
11	Segment of a multisegment record other than the first or last segment.

The SDW for the first segment replaces the RDW for the record after the record has been segmented. You or the operating system can build the SDW, depending on which access method is used.

For an illustration of variable spanned records, see Figure 58 on page 355.

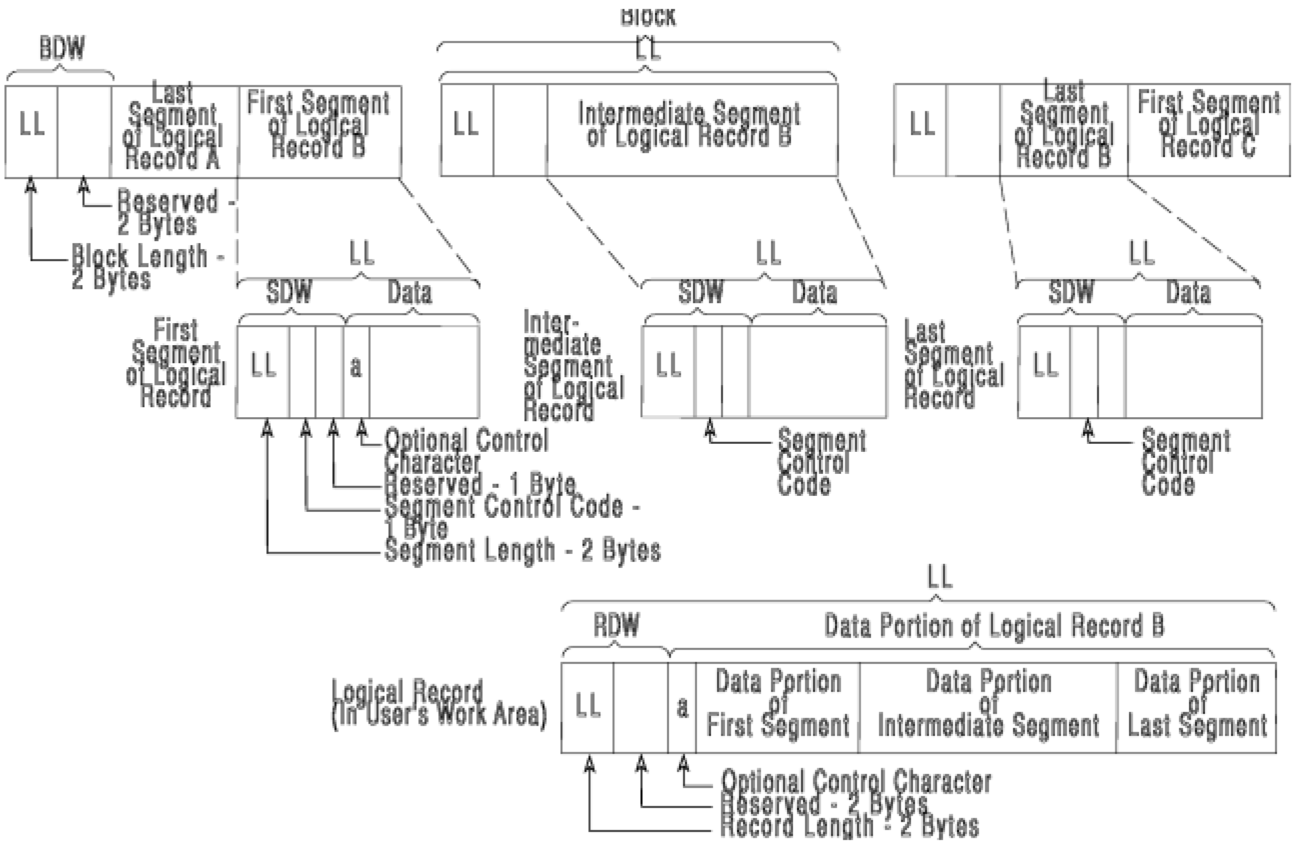

Block
LL
BDW
LL
Last Segment of Logical Record A
First Segment of Logical Record B
LL
Intermediate Segment of Logical Record B
LL
Last Segment of Logical Record B
First Segment of Logical Record C
Reserved - 2 Bytes
Block Length - 2 Bytes
LL
SDW
Data
First Segment of Logical Record
LL
a
Optional Control Character
Reserved - 1 Byte
Segment Control Code - 1 Byte
Segment Length - 2 Bytes
LL
SDW
Data
Intermediate Segment of Logical Record
LL
Segment Control Code
LL
SDW
Data
Last Segment of Logical Record
LL
Segment Control Code
LL
RDW
Data Portion of Logical Record B
Logical Record (In User's Work Area)
LL
a
Data Portion of First Segment
Data Portion of Intermediate Segment
Data Portion of Last Segment
Optional Control Character
Reserved - 2 Bytes
Record Length - 2 Bytes

Figure 58. Variable Spanned Records

For more information on variable spanned records, see the *MVS/XA Data Administration Guide*.

Variable Spanned ANSI Records

Variable spanned records that exist on ANSI tapes are recorded in 7-bit ASCII code. They are similar to those shown in Figure 58 on page 355 except that they contain a segment control word (SCW) instead of an SDW. The SCW is a 5-byte field that describes the record and is internally equivalent to the SDW. (See page "Segment descriptor word" on page 354 for more information on SDW.) Variable spanned ANSI records may also contain a block prefix. See Figure 59 on page 356 for an illustration.

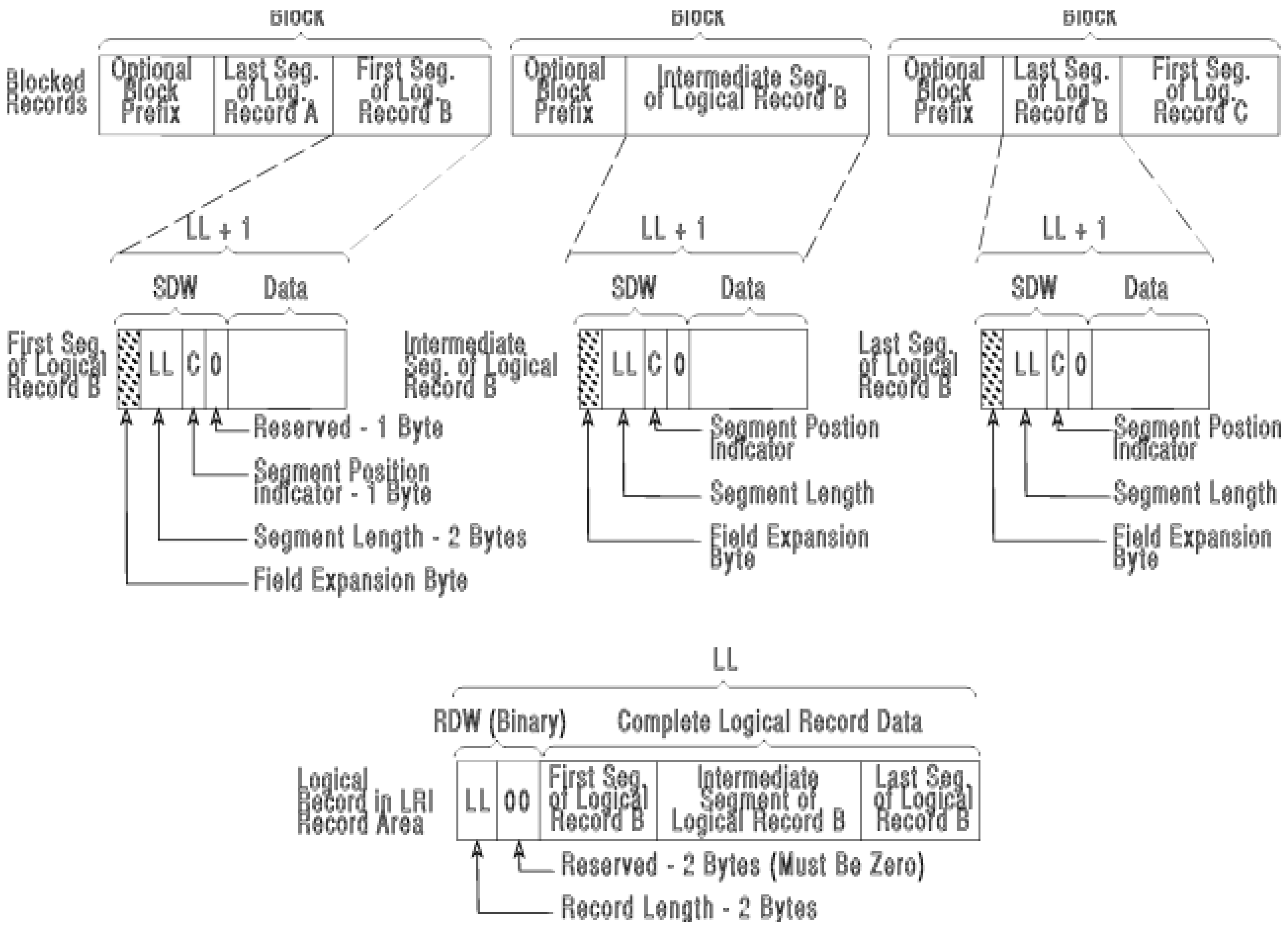

Block
Block
Block
Blocked Records
Optional Block Prefix
Last Seg. of Log. Record A
First Seg. of Log. Record B
Optional Block Prefix
Intermediate Seg. of Logical Record B
Optional Block Prefix
Last Seg. of Log. Record B
First Seg. of Log. Record C
LL + 1
LL + 1
LL + 1
SDW
Data
SDW
Data
SDW
Data
First Seg. of Logical Record B
LL
C
0
Reserved - 1 Byte
Segment Position indicator - 1 Byte
Segment Length - 2 Bytes
Field Expansion Byte
Intermediate Seg. of Logical Record B
LL
C
0
Segment Postion Indicator
Segment Length
Field Expansion Byte
Last Seg. of Logical Record B
LL
C
0
Segment Postion Indicator
Segment Length
Field Expansion Byte
LL
RDW (Binary)
Complete Logical Record Data
Logical Record in LRI Record Area
LL
00
First Seg. of Logical Record B
Intermediate Segment of Logical Record B
Last Seg. of Logical Record B
Reserved - 2 Bytes (Must Be Zero)
Record Length - 2 Bytes

Figure 59. Variable Spanned ANSI Records

To specify that a data set contains variable spanned ANSI tape records, use either the DS or DBS option of the RECFM parameter. RECFM can either be specified in the FILEDEF command or DCB macro. Support for variable spanned ANSI records is available through both QSAM and BSAM access methods.

The following example specifies variable spanned ANSI records to be read from a tape.

```
DCB DSORG=PS,DDNAME=ORANGE,MACRF=(GL),
    LRECL=100,BLKSIZE=1020,BUFOFF=20,RECFM=DBS,BFTEK=A
```

The BFTEK=A option specifies that the logical record interface will be used. For ANSI spanned records, RECFM=DS and RECFM=DBS is supported only through LRI in QSAM mode. For more information, see "Logical Record Interface" on page 376.

The BUFOFF parameter indicates a block prefix will be read. When using QSAM access method for input, the block prefix cannot be returned to the application. When writing to an output file using QSAM or BSAM, BUFOFF must be 0 or L. In CMS, you can only write a block prefix using L. For variable spanned records, using L specifies that you want to write a segment descriptor word (SDW).

Null Segments

A 1 in bit position 0 of the SDW indicates a null segment. A null segment means that there are no more segments in the block. Bits 1 to 7 of the SDW and the remainder of the block must be binary zeros. If you create a data set file using null segments, the results are unpredictable.

Moving Variable Spanned Records

Use the MOVEFILE command to move variable spanned records from any device supported by VM to any other device supported by VM. Note that when a CMS DASD device is used, the file containing the variable spanned records must have a file mode number of 4. For example,

```
FILEDEF IN TAP1 SL (RECFM VS
FILEDEF OUT DISK ARIZONA BBALL A4 (RECFM VS LRECL 1024
        BLOCK 512
MOVEFILE IN OUT
```

This moves the first file from the IBM standard labeled tape on TAP1 to the file ARIZONA BBALL A. The input record length and input block size for the IN file are taken from the tape label. The output file attributes on FILEDEF are used for the OUT file.

Error Handling for Spanned Records

If an error occurs during spanned record processing, error message 120 with error code 8 is issued.

Other errors include:

- If record segments of a variable spanned record are found to be in an improper sequence, unpredictable result can occur.
- If the record area is too small to contain a logical record, the GET or PUT will fail.

In either of these cases, if the SYNAD address is available, it will receive control and the error information can be obtained through the SYNADAF macro. If no SYNAD address is specified, CMS will abend.

For more information on SYNAD, see "Exit Routines" on page 387 and the *MVS/XA Data Administration Guide*.

When spanned records are stored on multiple volumes, errors may occur if a volume that begins with a middle or last segment is mounted first or if an FEOV macroinstruction is issued followed by another GET.

Identifying I/O Files and Devices to OS Simulation

For the application programmer, the most significant difference between the MVS and CMS support is how the input and output (I/O) files are defined to the system. In MVS, you set up the data definition (DD) statements in the OS/MVS job control language (JCL) to define the input and output datasets to the program. For example:

```
//MYIDX     JOB  (XYZ),'MYNAME',CLASS=A,MSGCLASS=A
//RUNIT     EXEC PGM=MYPROG
//MYINPUT   DD   DSN=MYID.INPUT.DATA,DISP=SHR
//MYOUTPUT DD    DSN=MYID.OUTPUT.DATA,DISP=OLD
```

In this example, MYPROG is the application program and MYINPUT is the name the program uses to refer to its input file. MYOUTPUT is the name the program uses to refer to its output file. To run this program, you would submit this JCL and the batch job would run in a separate initiator address space under jobname MYIDX.

Whenever you execute an OS/MVS program under CMS which has input or output files, you must first identify the files to CMS with the FILEDEF command. The FILEDEF command in CMS describes the input and output files. The FILEDEF command serves the same purpose as the JCL DD statement.

To run the above program under CMS OS simulation, you must define the I/O to the system using the FILEDEF command and execute the program using the CMS LOAD and START commands. For example:

```
FILEDEF MYINPUT  DISK INPUT  DATA A
FILEDEF MYOUTPUT DISK OUTPUT DATA A
LOAD    MYPROG    \
START             /  or simply LOAD  MYPROG (START
```

The program will then be executed in the CMS Virtual Machine's address space while the CMS user waits for it to complete.

When you enter the FILEDEF command, you specify:

- The *ddname*
- The device type
- A file identification, if the device type is DISK
- Type of label on your tape file, if tape label processing is specified
- Options (if necessary).

Some guidelines for entering these specifications follow.

Specifying the ddname

If the FILEDEF command is issued for a program input or output file, the *ddname* must be the same as the *ddname* or file name specified for the file in the source program. For example, you can have an assembler language source program that contains the line:

```
INFILE  DCB  DDNAME=INPUTDD,MACRF=(GL),
DSORG=PS,RECFM=F,LRECL=80
```

If you want to use the CMS file MYINPUT FILE A1 as your input file, you must issue a FILEDEF for this file before executing the program.

```
FILEDEF INPUTDD DISK MYINPUT FILE A1
```

When you want to read or write to the file, MYINPUT FILE A1, your program can refer to it using the *ddname*, INPUTDD. If the input file you want to use is on an OS/MVS disk accessed as your C-disk and it has a data set name of PAYROLL.RECORDS.AUGUST, then your FILEDEF command might be:

```
FILEDEF INPUTDD C1 DSN PAYROLL.RECORDS.AUGUST
```

The *ddname*, INPUTDD, is now associated with the OS/MVS file, PAYROLL.RECORDS.AUGUST.

Specifying the Device Type

The device type can be specified for 4 situations, reading input files, writing to output files, executing programs that you do not want real I/O to be performed, and clearing *ddnames*.

For input files

The device type you enter on the FILEDEF command indicates the device from which you want records read. It can be DISK, TERMINAL, READER (for input from real cards or virtual cards), or TAPn (for tape). Using the above example, if your input file is to be read from your virtual card reader, the FILEDEF command might be as follows:

```
filedef inputdd reader
```

Or, if you were reading from a tape attached to your virtual machine at virtual address 181 (TAP1):

```
filedef inputdd tap1
```

For output files

the device you specify can be DISK, PRINTER, PUNCH, TAPn (tape), or TERMINAL.

For DUMMY files

if you do not want any real I/O performed during the execution of a program for a disk input or output file, you can specify the device type as DUMMY. For example,

```
filedef inputdd dummy
```

For clearing existing ddname:

Use the CLEAR option. Clearing a *ddname* before defining it ensures that a file definition does not exist and that any options previously defined with the *ddname* no longer exist. For example,

```
filedef inputdd clear
```

Entering File Identifications

Using a CMS file

If you are using a CMS disk file for your input or output, specify:

```
filedef ddname disk filename filetype filemode
```

The file mode field is optional; if you do not specify it, your A-disk is assumed. If * is used for the file mode of an output file, unpredictable results may occur.

Using an OS/MVS simulated data set

If you want an output file to be constructed in OS/MVS simulated data set format, you must specify the file mode number as 4. For example, if a program contains a DCB for an output file with a *ddname* of OUTPUTDD and you are using it to create a CMS file named DAILY OUTPUT on your B-disk, specify:

```
filedef outputdd disk daily output b4
```

Using an OS/MVS data set

If your input file is an OS/MVS data set on an OS/MVS disk, you can identify it in several ways:

- If the data set name has only two qualifiers, for example HEALTH.RECORDS, you can specify:

  ```
  filedef inputdd disk health records b1
  ```

- If it has more than two qualifiers, you can use the DSN keyword and enter:

  ```
  filedef inputdd b1 dsn health records august 1990
                — or —
  filedef inputdd b1 dsn health.records.august.1990
  ```

  Or you can request a prompt for a complete data set name:

  ```
  filedef inputdd b1 dsn ?
  Enter data set name:
  health.records.august.1990
  ```

  Note: When you enter a data set name using the DSN keyword either with or without a request for prompting, you should omit the device type specification of DISK, unless you want to assign a CMS file identifier, as in the example below.

- You can also relate an OS/MVS data set name to a CMS file identifier:

```
filedef inputdd disk ossim file c1 dsn monthly records
                      — or —
filedef inputdd disk ossim file c1 dsn monthly.records
```

Then you can refer to the OS/MVS data set MONTHLY.RECORDS by using the CMS file identifier, OSSIM FILE:

```
state ossim file c
```

When you do not issue a FILEDEF command for a program input or output file or if you enter only the *ddname* and device type on the FILEDEF command then CMS issues a default file definition, as follows:

```
FILEDEF ddname DISK FILE ddname A1
```

where ddname is the *ddname* you assigned with the DCB macro in your program or on the FILEDEF command.

For example, if you assign a *ddname* of OSCAR to an output file and do not issue a FILEDEF command before you execute the program, then the CMS file FILE OSCAR A1 is created when you execute the program.

If the file type of a CMS input file, FILE *ddname* A1, is the same as the assigned *ddname*, the file can be identified by a default file definition. Even though an input file can be defined explicitly or by default, if an attempt is made to read the file and the file is not found, unpredictable results may occur.

Specifying CMS Tape Label Processing

You can use the label operands on the FILEDEF command to indicate the type of label processing to be done for your file. For example, to process a file on an ANSI labeled tape at the virtual address, 182, use the command,

```
FILEDEF MAGIC TAP2 AL VOLID LAKER
```

The following types of label processing can be performed by CMS:

- ANSI label and ANSI user label (AL/AUL)
- IBM standard label and IBM standard user label (SL/SUL)
- Nonstandard label (NSL)
- No label (NL)
- Bypass label processing (BLP)
- No CMS tape label processing (LABOFF).

CMS supports the MVS convention for writing standard labels in non-compacted mode when the data is actually written in 3480 compaction mode. This allows 3480 compacted tapes to be more portable between CMS and MVS systems. Open processing will check the HDR2 tape label **tape recording technique** field and the users requested (or defaulted) tape recording format code in the tape FILEDEF. If either indicates compaction, the tape drive must actually support the IDRC data compaction feature or else the open will fail. Message DMS115S will be issued if the FILEDEF tape format indicates compaction and the tape drive does not support the feature. Message DMS434E will be issued if the existing HDR2 label indicates data compaction and the tape drive does not support the feature.

If CMS OS Simulation is used to add data to an existing tape file using either the OPEN EXTEND or FILEDEF (DISP MOD processing, the new data must conform to the existing **tape recording technique** as specified in the HDR2 label. If it does not, OPEN processing will protect the integrity of the data by adjusting the user's recording format to match the recording format of the existing data.

For more information on CMS tape label processing, see "Tape Labels in CMS" on page 151.

Specifying Options

The FILEDEF command has many options; those mentioned below are a sampling only. For complete descriptions of all the options of the FILEDEF command, see the *z/VM: CMS Commands and Utilities Reference*.

Supplying File Format Information

If you are using the FILEDEF command to relate a data control block (DCB) in a program to an input or output file, you may need to supply some of the file format information on the FILEDEF command line, such as the block size (BLOCK or BLKSIZE), record length (LRECL), record format (RECFM), and data set organization (DSORG). For example, if you have coded a DCB macro for an output file as follows:

```
OUTFILE  DCB   DDNAME=OUT,MACRF=(PM),DSORG=PS
```

then, when you are issuing a FILEDEF for this *ddname*, you must specify the format of the file. To create an output file on disk blocked in OS/MVS simulated data set format, you could issue:

```
filedef out disk myoutput file a4
(recfm fb lrecl 80 block 1600
```

To punch the output file onto cards, you would issue:

```
filedef out punch (lrecl 80 recfm f
```

You can omit file format information on the FILEDEF command line whenever it is supplied on the DCB macro or whenever your file exists on an OS/MVS disk. For existing CMS disk files, format information is required only if you want OS-simulated data set formats other than F or V.

When the OPEN macro instruction is executed, the CMS simulation of the OS OPEN routine initializes the data control block (DCB). The DCB fields are filled in with information from the DCB macroinstruction, the information specified on the FILEDEF command, or if the data set already exists, the data set label. See "Filling in the DCB information" on page 382 for more information on DCB information.

Removing and Retaining File Definitions

To remove any existing definition for the specified *ddname*, use the CLEAR operand. Clearing a *ddname* before defining it ensures that a file definition does not already exist and that any options previously defined with the *ddname* no longer have effect.

Be careful not to clear a file definition after the corresponding DCB has been opened. The OS macros rely on information set up by the FILEDEF command. If this information is cleared, the macros are not able to function properly and unpredictable results can occur.

Note: Execs or programs should **never** issue the `FILEDEF * CLEAR` command because it will clear FILEDEFs which are needed by other programs. If those programs try to use the cleared FILEDEFs to open or close a file, errors will result.

Usually, when you execute one of the language processors, all existing file definitions are cleared. If the development of a program requires you to recompile and re-execute it frequently, you might want to use the PERM option when you issue file definitions for your input and output files. For example:

```
cp spool punch to *
filedef indd disk test file a1 (lrecl 80 perm
filedef outdd punch (lrecl 80 perm
```

In this example, because you spooled your virtual punch to your own virtual card reader, output files are placed in your virtual reader. You can either read or delete them.

All file definitions issued with the PERM option stay in effect until you log off, specifically clear those definitions, or redefine them:

```
filedef indd clear
filedef outdd tap1 (lrecl 80
```

In the above example, the definition for INDD is cleared; OUTDD is redefined as a tape file.

When you issue the command:

```
filedef * clear
```

all file definitions are cleared, except those you enter with the PERM option.

When a program abends, or when you issue the HX Immediate command, all file definitions are cleared, including those entered with the PERM option.

Adding Records to a File

When you issue a FILEDEF command for an output file and assign it a CMS file identifier that is identical with an existing CMS file, the existing file is replaced by the new output file if anything is written to that *ddname*. If you want, instead, to have new records added to the bottom of the existing file, you can use the DISP MOD option:

```
filedef outdd disk new update a1 (disp mod
```

The file must be on a disk accessed as read/write. Note that an extension of a disk is read-only.

You can also add records to the end of a file using the EXTEND parameter on the OPEN macro. For more information, see "Adding records to the end of an existing file" on page 379.

Specifying a Member Name of a Data Set

If the file you want to read is a member of an OS/MVS partitioned data set (or a CMS MACLIB or TXTLIB), you can use the MEMBER option to specify the member name. For example:

```
filedef test c dsn sys1.maclib (member test
```

defines the member TEST from the OS/MVS macro library SYS1.MACLIB.

Receiving Control during I/O Operation

The AUXPROC option of the FILEDEF command can be used to monitor and modify I/O operations. It requires some knowledge of the control blocks, OPSECT and FCB, in order to be used. See the *z/VM: CMS Macros and Functions Reference* for information on OPSECT and FCB.

Use the FILEDEF AUXPROC option to indicate the fullword address of an auxiliary processing routine you have written. The AUXPROC option is valid only when FILEDEF is executed by an internal program call. It is invalid when entered as a terminal command because it must specify an address. Your AUXPROC routine will receive control from CMS before any device I/O is performed. The CMS language interface programs use this feature for special I/O handling of certain (utility) data sets. If your AUXPROC routine issues an OS OPEN on a file, you must make sure that any changes to the file attributes do not interfere with any other file processing.

At the completion of your auxiliary routine processing, control returns to CMS, signaling whether I/O has been performed. If it has not been done, CMS performs the appropriate device I/O.

When the routine receives control from CMS, the general purpose registers contain the following information:

GPR2
= data control block (DCB) address

GPR3
= base register for CMS

GPR8
= CMS OPSECT address

GPR11
= file control block (FCB) address

GPR14
= return address in CMS

GPR15
= auxiliary processing routine address

all other registers
= work registers

The FCBSECT can be found in the DMSOM MACLIB as member CMSCB. The OPSECT can be found in the DMSOM MACLIB as member IO. See the *z/VM: CMS Macros and Functions Reference* for information about OPSECT and FCBSECT.

The auxiliary processing routine must provide a save area to save the general purpose registers. This routine must also perform the save operation. CMS does not provide the address of a save area in general purpose register 13, as is usually the case. When control returns to CMS, the general purpose registers must be restored to their original values. Control is returned to CMS by branching to the address contained in general purpose register 14.

GPR15 is used by the auxiliary processing routine to inform CMS of the action that has been or should be taken with the data block as follows:

Register Content	**Action**
GPR15=0	No I/O performed by AUXPROC routine. CMS performs I/O.
GPR15<0	I/O performed by AUXPROC routine and error was encountered. CMS takes error action.
GPR15>0 and GPR15<X'10000'	I/O performed by AUXPROC routine with residual count in GPR15.
GPR15=X'10000'	I/O performed by AUXPROC routine with zero residual count.

GPR15 should not be returned with a value greater than X'10000'. If it does, the system will treat the last two bytes of GPR15 as the residual count.

Passing Information to the DMSTVI Routine

An interface routine, DMSTVI, can be used to give control to a different multivolume switching routine than the one supplied with VM or to a tape management system. Use the SYSPARM option to pass information not included on the FILEDEF or LABELDEF command to the DMSTVI routine. See "Passing Information to the DMSTVI Routine" on page 399 for more information on SYSPARM and the DMSTVI routine.

Using OS/MVS Simulated Data Sets in CMS

The CMS access method support allows you to store your OS/MVS data on DASD in two different formats:

1. CMS format - like any CMS minidisk file
2. OS/MVS simulated format - file mode number 4

In general, OS/MVS simulation uses the CMS file formats to simulate different types of OS/MVS files. For more information on the OS record formats, see the *MVS/XA Data Administration Guide*.

OS Record Format	CMS File Format for Simulation
F Fixed, Unblocked **FB** Fixed, Blocked **FBS** Fixed, Standard Block	**F** Fixed format
V Variable, Unblocked **VB** Variable, Blocked **VS** Variable, Spanned **VBS** Variable, Blocked, Spanned **U** Undefined format	**V** Variable format

As long as the file mode number specified on the FILEDEF command for your output file is not 4, OS/MVS simulation will write your data to the DASD device in standard CMS format. A file mode number 4 indicates that the file is to be written in OS/MVS simulated format.

Storing OS/MVS Data in CMS Format DASD Files

CMS format files created by OS/MVS simulation have the following attributes:

- Each logical OS record is written as a separate record (line) in the CMS file, even if a blocked OS record format is used.
- For variable format files (V and VB), the 4-byte block descriptor word (BDW) and each 4-byte record descriptor word (RDW) is deleted before the records are written. On input, OS/MVS simulation restores the descriptor words for V format records inside the virtual storage I/O buffers. The stripping and restoring of descriptor information makes the OS created files compatible with CMS created files and CMS utilities.
- The CMS record length for fixed format files is equal to the LRECL defined in the file's DCB. For variable format files, the CMS record length is equal to the length of the longest record in the file.
- For fixed format files (F, FB, and FBS), the file must be referenced with the same LRECL defined in the file's DCB.

Specifying CMS Data Files

To illustrate how OS/MVS simulation creates CMS format files, assume that you have defined your output file using the following FILEDEF command;

```
FILEDEF OUTFILE DISK CMSFORM FILE A1 (LRECL 20 BLKSIZE 60 RECFM FB
```

This requests a CMS format file (A1) that contains fixed length records of 20 bytes (LRECL 20, RECFM FB). Let's say your application writes five records to this file and then closes it. If you were to XEDIT the file, you would see the following:

```
CMSFORM   FILE      A1   F 60   Trunc=60 Size=2 Line=1
           Col=1 Alt=10
===== * * * Top of File * * *
===== Record 1 01234567890
===== Record 2 01234567890
===== Record 3 01234567890
===== Record 4 01234567890
===== Record 5 01234567890
===== * * * End of File * * *
```

Note that although blocking was requested in the FILEDEF command (LRECL 20 BLKSIZE 60), there is no record blocking in the output CMS file. Each OS record is written as a separate CMS record.

Blocking for CMS Format Files

Here are some concepts you should know about blocking for CMS format files under OS/MVS simulation:

- Using a blocked record format (FB, FBS, VB, VBS) reduces the amount of CMS file system overhead associated with OS/MVS data management requests by reducing the number of file system requests made by OS/MVS simulation.

  With blocking, OS/MVS simulation can read or write several CMS records at a time. The number of records read or written is the number of records in one OS/MVS block. However, each logical OS/MVS record is still recorded as a separate CMS record on the DASD device.

  Without blocking, OS/MVS simulation must call the file system once for every record in the file.
- Under QSAM simulation, CMS does all of the blocking and deblocking for you, so the application does not have to worry about where records begin and end in its I/O buffers. For FB format files, QSAM simulation will only write a short block for the very last block in the file.
- Under BSAM and BPAM simulation, it is the application's responsibility to block and deblock records in the I/O buffers. If you write a short block (by reducing the value of the DCBBLKSI field in the DCB), all subsequent blocks will be written using the shorter block size, even if you restore the original DCBBLKSI value after the "short write". Additionally, on input, CMS uses the LRECL and BLKSIZE information specified for the input file to perform any blocking. CMS has no way of knowing what block sizes were used to write the file to the device.

Storing OS/MVS Data in OS/MVS Simulated DASD Files

For some of your OS/MVS applications, you may want to store your data in OS/MVS format; that is, actually write blocked data to the device in physical blocks, rather than breaking it up into logical records first. For these applications, CMS provides OS/MVS simulated files.

OS/MVS simulated files are maintained on CMS disks but in OS format rather than in CMS format. Because they are CMS files, you can edit, rename, copy, or manipulate them just as you would any other CMS file. These files are created and identified by specifying a file mode number of 4 on the FILEDEF command that describes the file.

By simulating OS/MVS macros, CMS simulates the following access methods so that OS/MVS data organized by these access methods can reside on CMS disks:

- BDAM
- BPAM
- BSAM/QSAM
- VSAM.

See “Accessing Data through OS/MVS Simulation” on page 373 for more information on these access methods.

Because CMS does not simulate the indexed sequential access method (ISAM), no OS/MVS program using ISAM can execute under CMS. Therefore, no program can write an indexed sequential data set on a CMS disk.

OS/MVS simulated files have the following attributes:

- Each block of information is written to DASD intact as a single record in the CMS file.
- For fixed format files (F, FB, and FBS), the file must have the same LRECL and BLKSIZE in the DCB as the file you are referencing. One exception to this is when you are writing FB or FBS format files under BSAM or BPAM simulation. For this case, you may specify a BLKSIZE value that is less than the BLKSIZE of the file you are referencing.
- For variable format file (V and VB), all block and record descriptor words are written to the file as part of the block (just as it would be by an OS or MVS system).
- The CMS record length for the file is equal to the length of the largest block in the file (for fixed format files, all blocks are the same size).
- OS simulated blocks consist of OS simulated records. The term OS simulated block corresponds to the CMS record. There is no equivalent term in CMS for the OS simulated record. See Figure 60 on page 366 for clarification.

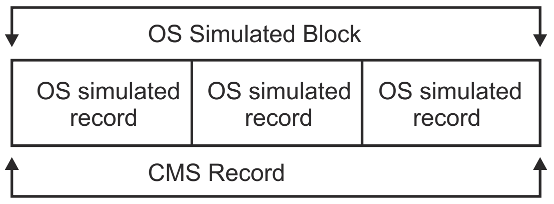


Figure 60. OS Simulated Block vs. CMS Record

Note: CMS does not support the writing of short blocks anywhere other than to the last block of a file. Although such a request will not be prevented, any later attempt to read from or write to the file may produce unpredictable results.

Specifying OS/MVS Simulated Data Files

Let us examine how the previous example ("Specifying CMS Data Files" on page 364) would work if we used an OS/MVS simulated file. In this case, the FILEDEF command would look like this:

```
FILEDEF OUTFILE DISK OSFORM FILE A4
             (LRECL 20 BLKSIZE 60 RECFM FB
```

Note that the only difference between this example and the one in "Specifying CMS Data Files" on page 364 is the file name and the file mode number. For OS/MVS simulated data file, you must specify a file mode number of 4.

```
OSFORM    FILE      A4   F 60   Trunc=60 Size=2
          Line=1 Col=1 Alt=10
===== * * * Top of File * * *
===== Record 1 01234567890Record 2 01234567890Record 3
             01234567890
===== Record 4 01234567890Record 5 01234567890/
             /
===== * * * End of File * * *
```

Each CMS record now contains one OS block. Notice the characters following record 5. The first 4 characters are the end of the file marker; this is the character string X'61FFFF61'. The X'61' is displayed as a '/'. Because X'FF' is a nondisplayable character, XEDIT substitutes a double quotation mark. This is also the case with the rest of the nondisplayable characters in the block.

End-of-file marker

OS/MVS simulation uses the end-of-file (EOF) marker (X'61FFFF61') to identify the end of the OS data in a file when all of the following conditions are met:

- File mode number is 4
- OS record format is FB or FBS
- Last block of the file is a short block

- File is being processed under the QSAM, BSAM, or BPAM access method.

Because CMS file system will not allow you to write records of varying length to a fixed format CMS file, the EOF marker marks the end of a file.

There are a few other places where the EOF marker is used:

- Any time the MOVEFILE command is being used to read a file.
- Any time a partitioned data set (PDS) is being read by OS/MVS simulation. A file is considered to be a PDS when the DCB data set organization is specified as partitioned (DSORG=PO), when the MEMBER option of the FILEDEF command is specified for the file, or when the PDS option has been specified on the MOVEFILE command that is reading the file.

Note: When writing an FB or FBS format file, under BSAM or BPAM, you may specify a BLKSIZE value that is less than the file you are referencing. This is done to indicate that the last block is a short block, causing OS/MVS simulation to place an end-of-file marker (described in "End-of-file marker" on page 366) after the last logical record in the output block.

One last example, let's see what a VB format file would look like. In the FILEDEF command:

```
FILEDEF OUTFILE DISK OSFORM VFILE A4
         (LRECL 70 BLKSIZE 74 RECFM VB
```

the OSFORM VFILE will have variable blocked records (RECFM VB). The maximum length of a record will be 66 bytes plus 4 bytes for a record descriptor word (LRECL 70). Each block must be large enough to hold the longest record (LRECL 70) plus a 4-byte block descriptor word (BLKSIZE 74). Now if your application writes 5 records of lengths 20, 8, 15, 37, and 15 bytes (respectively) to a file, closes it, and you were to XEDIT the file, you would see the following:

```
OSFORM    FILE      A4   F 64   Trunc=64 Size=2
          Line=1 Col=1 Alt=10
===== * * * Top of File * * *
===== Record 1 01234567890 Record 2
          Record 3 012345
===== Record 4 0123456789012345678901234567
          Record 5 012345
===== * * * End of File * * *
```

The first four nondisplayable characters on each line are block descriptor words. The four nondisplayable characters in front of each record are the record descriptor words for those records. Note that the CMS record length for the file is shown as 64; this is the length of the longest block in the file.

Considerations for Files in Shared File System Directories

OS/MVS data can be stored in a Shared File System directory and manipulated through OS simulation. Here are some special considerations to make note of when updating OS/MVS data that resides on shared file directories:

- As long as no read/write sharing is taking place (for example, one user reads a file while another user writes to it), the version of all data sets is guaranteed to remain constant to the OS application. Only those changes made by the application itself will be seen. This guarantee does not exist for a read/write sharing environment.
- All shared file directories that are used for output by an OS application must be accessed as R/W.
- Unpredictable results may occur if the default SFS work unit is changed during the execution of a program that uses OS simulation. Also, if a file used by an OS simulation application is already open (through FSOPEN, EXECIO, or a previous OS OPEN) at entry to that application and the default work unit on which the file was opened is different from that used during the application, unpredictable results may occur if the application closes and then reopens the file.
- If an application opens a data set whose FILEDEF specifies a file mode of * for output, OS simulation will search all R/W file modes for the file. If it finds the file on any of those file modes, it will use the first occurrence of the file. Otherwise, OS simulation will create a new file with mode A1 when the application does the first write to the file.

- Each time a CLOSE is issued for a DASD DCB without the TYPE=T parameter, the file represented by that DCB will have FINIS issued for it.
- When you execute programs with OS macros that write to SFS files, you should ensure that there is an ample supply of unused blocks in the file space to which you are writing. Otherwise, your program could abnormally end when your program tries to close the file. You might also consider limiting write access to the file space so that other users do not consume the blocks you require for your application.
- If your application uses OS/MVS queued sequential access method (QSAM) to update SFS files, be sure to erase all data from the buffers (for example, close the files) before issuing a commit. This may be required because of QSAM buffering. For example, if a program uses OS/MVS QSAM with blocked records for an SFS output file, some of the most recently written output records may not be committed if the program issues a commit without first closing the QSAM file. These records will subsequently be committed when the program closes the file and either commits the work unit or allows end-of-command processing to commit the work unit. Using only FS macros to write to SFS files avoids this situation.
- An OS Simulation WRITE/PUT to a file in an SFS server will generally receive a disk full error when the SFS file space limit is exceeded. However, an OS Simulation WRITE/PUT to a file in an SFS server running at release levels prior to Release 2.1 will detect a disk full condition when the file space threshold is exceeded.

Using OS/MVS Data Sets in CMS

By simulating OS/MVS macros, CMS can read sequential and partitioned data sets that reside on OS/MVS disks. However, a sequential or partitioned data set that resides on an OS/MVS disk can only be written or updated by an OS/MVS program running in an OS system. CMS cannot write or update data sets on OS/MVS disks.

CMS can execute programs that read and write VSAM files from OS/MVS programs written in the VS BASIC, COBOL, PL/I, VS/APL, and VS FORTRAN programming languages. CMS also supports VSAM for use with DOS/VS SORT/MERGE. This CMS support is based on the VSE/VSAM licensed program and, therefore, the OS/MVS user is limited to those VSAM functions that are available under VSE/VSAM.

CMS Commands You Can Use with OS/MVS Data Sets

The CMS commands that recognize OS/MVS data sets on OS/MVS disks are listed in Table 36 on page 368.

Table 36. CMS Commands that Recognize OS/MVS Data Sets on OS/MVS Disks

Command	Operation
ACCESS	Makes the OS/MVS disk containing the data set available to your CMS virtual machine.
ASSEMBLE	Assembles an OS/MVS source program under CMS.
DDR	Copies an entire OS/MVS disk to tape.
FILEDEF	Defines the OS/MVS data set for use under CMS by associating an OS/MVS *ddname* with an OS/MVS data set name. Once defined, the data set can be used by an OS/MVS program running under CMS and can be manipulated by the other commands that support OS/MVS functions.
GLOBAL	Makes macro libraries or LOADLIB libraries available to CMS. You can prepare an OS/MVS library for reference by the GLOBAL command by issuing a FILEDEF command for the data set and giving the data set the appropriate file type of MACLIB or LOADLIB.
LKED	Creates CMS LOADLIB libraries from CMS TEXT files or OS object modules.
LISTDS	Lists information describing OS/MVS data sets residing on OS/MVS disks.

Table 36. CMS Commands that Recognize OS/MVS Data Sets on OS/MVS Disks (continued)

Command	**Operation**
MOVEFILE	Moves data records from one device to another device. Each device is specified by a *ddname*, which must have been defined by FILEDEF. You can use the MOVEFILE command to create CMS files from OS/MVS data sets.
NUCXLOAD	Loads, relocates, and establishes as a nucleus extension a load module either from a CMS LOADLIB or from an OS/MVS module library on an OS/MVS formatted disk.
OSRUN	Loads, relocates, and executes a load module either from a CMS LOADLIB or from an OS/MVS module library on an OS/MVS formatted disk.
QUERY	Lists (1) the files that have been defined with the FILEDEF and DLBL commands (QUERY FILEDEF, QUERY DLBL), or (2) the status of OS/MVS disks attached to your virtual machine (QUERY DISK, QUERY SEARCH).
RELEASE	Releases an OS/MVS disk you have accessed (by ACCESS) from your CMS virtual machine.
STATE	Verifies the existence of an OS/MVS data set on a disk. Before STATE can verify the existence of the data set, you must have defined it (using FILEDEF).

Accessing OS/MVS Data Sets

Before CMS can read an OS/MVS data set that resides on a non-CMS disk, you must issue the CMS ACCESS command to make the disk available to CMS. You must not specify options or file identification when accessing an OS/MVS disk. For more details, see the *z/VM: CMS User's Guide*.

Using OS Format Disks on CMS

MVS and DOS (OS) format disks which may be resident volumes on an actual MVS or DOS system may be accessed by CMS as a Read-Only minidisk. The organization of these disks is very different from that of a CMS minidisk, so ordinary CMS commands will not work on OS format disks. However, certain CMS commands can use files residing on OS format disks if the files are defined to CMS with a FILEDEF command.

Listing Information about OS Disks with the LISTDS Command

The QUERY DISK command gives usage information about normal CMS minidisks, but gives minimal information about OS disks. You can use the LISTDS command with the FREE option to obtain information about free space on an OS disk. For example:

```
listds n (free
Freespace extents for N disk:
  CYL-HEAD    (RELTRK) TO   CYL-HEAD    (RELTRK)     TRACKS
00000 00008          8    00001 00014         29         22
Ready;
```

The LISTDS command with the PDS option provides information about the libraries residing on an OS disk.

The LISTFILE command gives information about minidisk or SFS files, but not about files on an OS disk. You can use the LISTDS command with the FORMAT option to obtain information about OS disk files. For example:

```
listds n (format
RECFM LRECL BLKSI DSORG   DATE   LABEL  FM DATA SET NAME
FB    80    27920 PO    12/12/91 MVS292 N  MY.OWN.MACLIB
Ready;

listds n (extent

Extent information for VTOC on N disk:
SEQ TYPE    CYL-HEAD    (RELTRK) TO   CYL-HEAD    (RELTRK)     TRACKS
```

```
000 VTOC  00000 00001          1    00000 00002          2          2

Extent information for MY.OWN.MACLIB on N disk:
SEQ TYPE    CYL-HEAD    (RELTRK) TO   CYL-HEAD    (RELTRK)     TRACKS
000 DATA  00000 00003          3    00000 00007          7          5
Ready;
```

Using the GLOBAL Command with OS Files

The GLOBAL command can use OS files if they are defined with the FILEDEF command with a name normally used in the global list. The FILEDEF command equates an OS data set name such as A.B.C with a CMS minidisk *fn ft fm* such as MY DOSLIB A1. Any *ddname* may be used, but the device must be DISK. *fn* must be one of the names from the GLOBAL list and *ft* must match the type of GLOBAL command, such as a file type of LOADLIB for a GLOBAL LOADLIB command. *fm* must be the file mode used to access the OS disk. The *dsn* value must be the actual OS data set name. For example:

```
listds f
FM DATA SET NAME
F  MY.OWN.MACLIB
Ready;

filedef syslib disk os maclib f dsn my.own.maclib
Ready;

query filedef
SYSLIB   DISK     OS       MACLIB   F1  MY.OWN.MACLIB
Ready;

global maclib os
Ready;

query maclib
MACLIB   = OS
Ready;
```

Using XEDIT with OS Files

XEDIT can use OS files if they are defined with the FILEDEF command with a *ddname* of SYSIN. Most XEDIT commands can be used on an OS file, including using the XEDIT PUT command and filing on a different file mode. However, since OS disks are Read-Only to CMS, XEDIT may not be used to file an OS data set back onto the OS disk. The *ddname* must be SYSIN, and the device type must be DISK. *fn* and *ft* may be any valid CMS names. *fm* must be the file mode used to access the OS disk. The *dsn* value must be the actual OS data set name. For example:

```
Ready;

acc 291 f
DMSACC723I F (0291) R/O - OS
Ready;

listds f
FM DATA SET NAME
F  MY.OWN.MACLIB
F  MY.OWN.FMTFXD
F  MY.OWN.FMTVAR
Ready;

filedef sysin disk os file f dsn my.own.fmtfxd
Ready;

xedit os file f
```

Creating CMS Files from OS/MVS Data Sets

If you have data sets on OS/MVS disks, tapes, or cards, you can copy them into CMS files so that you can edit, modify, or manipulate them with CMS commands. The CMS MOVEFILE command copies OS/MVS (or CMS) files from one device to another. You can move data sets from any valid input device to any valid output device.

Before using the MOVEFILE command, you must define the input and output data sets or files and assign them *ddnames* using the FILEDEF command. If you use the *ddnames* INMOVE and OUTMOVE, then you do not need to specify the *ddnames* when you issue the MOVEFILE command. For example, the following sequence of commands copies a OS/MVS disk file into your virtual card punch:

```
filedef inmove disk diskin file a1
filedef outmove punch
movefile
```

The result of these commands is effectively the same as if you had issued the command:

```
punch diskin file (noheader
```

This example illustrates the basic relationship between the FILEDEF and MOVEFILE commands. In addition to the MOVEFILE command, if the OS/MVS input data set is on tape or cards, you can use the TAPPDS or READCARD command to create CMS files.

When you copy a variable-length data set or variable spanned data set from an OS/MVS disk to a CMS disk, the logical record length (LRECL) of the file that is created on the CMS disk is equal to the size of the largest record in the data set being copied. If the file that is being created has a file mode of 4, the logical record length is equal to the LRECL of the largest record plus 8 bytes. The actual LRECL of the new file can be determined by using the CMS FILELIST command.

Copying Sequential Data Sets from Disk

The MOVEFILE command can copy a sequential OS/MVS disk data set from a read-only OS/MVS disk into an integral CMS file on a CMS read/write disk. You use FILEDEF commands to identify the input file disk mode and data set name:

```
filedef inmove c1 dsn sales.manual
```

the CMS output file's disk location and fileid:

```
filedef outmove disk sales manual a1
```

and then you issue the MOVEFILE command:

```
movefile
```

Copying Partitioned Data Sets from Disk

The MOVEFILE command can copy partitioned data sets (PDS) into CMS disk files and create separate CMS files for each member of the data set. You can have the entire data set copied, or you can copy only a selected member. For example, if you have a partitioned data set named ASSEMBLE.SOURCE whose members are individual assembler language source files, your input file definition might be:

```
filedef inmove c1 dsn assemble source
            — or —
filedef inmove c1 dsn assemble.source
```

If ASSEMBLE.SOURCE has 3 members, LIONS, TIGERS, BEARS, you could create individual CMS ASSEMBLE files, by issuing the output file definition as:

```
filedef outmove disk zoo assemble a1
```

Then use the PDS option of the MOVEFILE command:

```
movefile (pds
```

When the CMS files are created, the member names are used for the file name and the file type on the output file definition is used for the file type. For the previous example, 3 files would be created, LIONS ASSEMBLE, TIGERS ASSEMBLE, and BEARS ASSEMBLE.

If you want to copy only a single member, you can use the MEMBER option of the FILEDEF command:

```
filedef inmove disk assemble source c (member lions
```

and omit the PDS option on the MOVEFILE command:

```
movefile
```

Only the file LIONS ASSEMBLE would be created.

Summary

The following tables summarize the various ways that you can create CMS files from OS/MVS data sets.

Table 37. Input File. An OS sequential data set named: COMPUTE.TEST.RECORDS

Source	CMS Command Examples	CMS Output File
Disk: OS R/O C-disk	filedef indd c1 dsn compute test records filedef outdd disk compute records a1 movefile indd outdd	COMPUTE RECORDS A1
Tape: 181	filedef inmove tap1 (lrecl 80 filedef outmove disk test records a1 movefile tappds newtest compute (nopds	TEST RECORDS A1 NEWTEST COMPUTE A1
Cards:	filedef cardin reader filedef diskout disk compute cards a1 movefile cardin diskout readcard compute test	 COMPUTE CARDS A1 COMPUTE TEST A1

Table 38. Input File. OS partitioned data set named: TEST.CASES; members named: SIMPLE, COMPLEX, MIXED

Source	CMS Command Examples	CMS Output File
Disk: OS R/O C-disk	• filedef infile c1 dsn test cases • filedef outfile disk new testcase a1 • movefile infile outfile (pds • filedef in c1 dsn test cases (member simple • filedef run disk • movefile in run	• SIMPLE TESTCASE A1 • COMPLEX TESTCASE A1 • MIXED TESTCASE A1 • FILE RUN A1
Tape: 182	tappds * testrun (tap2	SIMPLE TESTRUN A1 COMPLEX TESTRUN A1 MIXED TESTRUN A1

Accessing Data through OS/MVS Simulation

An access method governs the manipulation of data. CMS can access files on both CMS disks and OS/MVS disks by simulating OS/MVS macros and access methods.

To execute OS/MVS code under CMS, the processing program must see data as OS/MVS would present it. For instance, when the processors expect an access method to acquire input source cards sequentially, CMS invokes specially written routines that simulate the OS/MVS sequential access method and pass data to the processors in the format that the OS/MVS access methods would have produced. Therefore, data appears in storage as if it had been manipulated using an OS/MVS access method. Block descriptor words (BDW), buffer pool management, and variable records are updated in storage as if an OS/MVS access method had processed the data. However, the actual writing to and reading from the I/O device is handled by the CMS file system.

The OS simulated access methods use the master file directory (MFD) and the file status table (FST) for information to access data. (In OS, this information is found in the volume table of contents and the data set control block.) A MFD updates the disk contents, and a FST describes each data file. All CMS disks are formatted in physical blocks of 512, 1024, 2048, or 4096 bytes.

To accomplish OS/MVS simulation, CMS supports the following access methods:

BDAM
(direct) - identifying a record by a key or by its relative position within the data set.

BPAM
(partitioned) - seeking a named member within data set.

Note: Two BPAM files with the same file type cannot be updated at the same time.

BSAM/QSAM
(sequential) - accessing a record in a sequence in relation to preceding or following records.

VSAM
(direct or sequential) - accessing a record sequentially or directly by key or address.

Note: CMS support of OS/MVS VSAM files is based on VSE/VSAM. Therefore, the OS/MVS user is restricted to those functions available under VSE/VSAM. See "OS/VSAM Macros Supported in CMS" on page 475 for more information.

CMS also updates those portions of the OS/MVS control blocks needed by the OS/MVS simulation routines to support a program during execution. The CMSCVT macro invokes the CVTSECT control block and simulates the communication vector table. Location X'10' contains the address of the CVT control section.

Note: The MVS version of the CVT is now included in OSMACRO1 MACLIB.

Accessing Data with OS/MVS Macros

Requests for input and output can be accomplished through several OS simulated macros. These macros support fixed and variable length records. Also, variable spanned records are supported through QSAM and BSAM.

GET (QSAM)
GET LOCATE and GET MOVE are supported.

GET (QISAM)
QISAM is not supported in CMS.

PUT (QSAM)
PUT LOCATE and PUT MOVE are supported. When you use Locate mode, issue an explicit CLOSE prior to returning to CMS to obtain the last record written with a PUT macro.

PUT (QISAM)
QISAM is not supported in CMS.

PUTX(QSAM)
PUTX support is provided only for data sets OPENed for Update with simple buffering.

READ/WRITE (BISAM)
BISAM is not supported in CMS.

READ/WRITE (BSAM and BPAM)
All the BSAM and BPAM options of READ and WRITE are supported except for the SB option (read backward).

READ (Offset Read of Keyed BDAM data set)
This type of READ is not supported because it is used only for spanned records.

READ/WRITE (BDAM)
All the BDAM and BSAM (create) options of READ and WRITE are supported except for the R and RU options.

BSP
When backspacing a non-OS simulated file, BSP assumes that the blocking factor of the file is the blocking factor used on the most recent write operation. Based on this value, the file is positioned to the previous blocking boundary.

When an input or output error occurs, do not depend on OS/MVS sense bytes. An error code is supplied by CMS in the ECB in place of the sense bytes. These error codes differ for various types of devices. Their meaning in z/VM can be found in *z/VM: CMS and REXX/VM Messages and Codes* under DMS message 120S.

For a disk device, when an input or output error occurs and a SYNAD routine is not specified, message 120S is issued and CMS is terminated abnormally during READ/WRITE processing.

Special Considerations for Blocked Data

Some special considerations must be made when using blocked data OS/MVS macros with CMS filemodes other than filemode 4. OS Simulation supports both Fixed Blocked (FB) and Variable Blocked (VB) data records when CMS filemode 4 is used, so that most common OS/MVS macros that use blocked data yield similar results under both the OS/MVS and VM operating systems.

Fixed Blocked Files

CMS filemode 1 files do not have blocked characteristics. When CMS filemode 1 files are used with OS Simulation for blocked record reads and writes, the records are treated as individual units rather than as blocked groups. This causes some problems when OS/MVS block record macro functions, such as READ, WRITE, NOTE, POINT, and BSP (BackSPace), are used on filemode 1 records. Normally these functions return a reference to the block number of a record grouping. However, when CMS filemode 1 files are used with these functions, the records must be logically pseudo-blocked because they exist as individual pieces rather than blocked groups. For Fixed Blocked records, both the size of each record and the size of the block is known, so the OS Simulation code can do a simple calculation to determine the OS/MVS block number reference based on the CMS relative record number, the record length and the block size. Therefore, Fixed Blocked files stored as CMS filemode 1 types will get the same block reference values returned as those stored as filemode 4 types.

Variable Blocked Files

The results for Variable Blocked files under CMS filemode 1 processing are different. Because the records are stored as individual pieces in common CMS file format, there are no Block Descriptor Word (BDW) or Record Descriptor Word (RDW) lengths that are kept with the data as when using filemode 4. Therefore, the OS Simulation code cannot calculate a block reference number to be associated with any particular record in the filemode 1 file. This means that OS/MVS blocked record macro functions cannot return a real OS block reference number. Instead, functions like NOTE and BSP return the real CMS filemode 1 relative record number for the last record that was processed. If application code is imported from an OS/MVS environment and uses blocked reference macros on Variable Blocked files, the files should always be be defined as CMS filemode 4 types. Otherwise, unpredictable results may occur if the application tries to make use of the CMS relative record number as if it were a real OS/MVS block reference number.

BDAM Restrictions

The four methods of accessing BDAM records are:

1. Relative Block RRR
2. Relative Track TTR
3. Relative Track and Key TTK
4. Actual Address MBBCCHHR.

The restrictions on these access methods are as follows:

- Only the BDAM identifiers underlined above can be used to refer to records because the CMS simulation of BDAM files uses a three-byte record identifier.
- CMS BDAM files are always created with 255 records on the first logical track and 256 records on all other logical tracks, regardless of the block size. If BDAM methods 2, 3, or 4 are used and the RECFM is U or V, the BDAM user must either write 255 records on the first track and 256 records on every track thereafter, or the BDAM user must not update the track indicator until a NO SPACE FOUND message is returned on a write. For method 3 (WRITE ADD), this message occurs when no more dummy records can be found on a WRITE request. For methods 2 and 4, this does not occur and the track indicator is updated only when the record indicator reaches 256 and overflows into the track indicator.
- Two files of the same file type, both using keys, cannot be open at the same time. If a program that is updating keys does not close the file it is updating for some reason, such as a system failure or another IPL operation, the original keys for files that are not fixed format are saved in a temporary file with the same file type and a file name of $KEYSAVE. To finish the update, run the program again.
- Variable-length BDAM files must be created under CMS in their entirety, with the XTENT option of FILEDEF specifying the exact number of records to be written. When reading variable BDAM files, the XTENT and key length information specified must duplicate what was created at file creation time. CMS does not support adding variable-length records to BDAM files.
- After a file is created using keys, additions to the file must not be made without using keys and specifying the original length.
- Note that there is limited support from the CMS file system for BDAM created files (sparse). Sparse files are manipulated with CMS commands but are not treated as sparse files by most CMS commands. The number of records in the FST is treated as a valid record number.
- The number of records in the data set extent must be specified using the FILEDEF command. The default size is 50 records.
- The minimum LRECL for a CMS BDAM file with keys is eight bytes.

Using the OS/MVS Simulated Buffering Techniques

OS simulation uses buffers to assemble and disassemble the input and output records of the application. These buffers belong to a specific DCB.

Obtaining I/O Buffers

The system will obtain the necessary buffers, but the application may obtain its own buffers with the GETPOOL macro. The application may issue the GETPOOL macro either before it opens the DCB or in the DCB Open exit. If the buffers supplied to Open processing are too small, the Open will fail with error code 35. Any application which obtains its own buffers MUST issue a FREEPOOL macro after it closes the DCB using the buffers. This frees the buffers which the GETPOOL macro obtained.

Determining the Minimum Buffer Size

In most cases, the blocksize is the minimum buffer size. The buffer must be large enough to hold the data which will be written out. Any BPAM, BDAM, or BSAM I/O will be done in blocks. Any Blocked or Spanned I/O will be done in blocks. If QSAM I/O (Get/Put) is specified and the record format is neither blocked nor spanned (that is F or V, not FB/VB/VS/VBS), then the minimum size for Fixed record format is the logical record length and the minimum size for Variable record format is the logical record length plus four bytes for the Record Descriptor Word (RDW). For any ANSI tapes, the Buffer Offset value (DCBBUFOF) must be added to the above minimum values.

Segment Interface

If the logical record interface is not specified, the segment interface is the default buffer technique for OS simulation. For input records, the application is responsible for assembling record segments into logical records by issuing GET or READ macroinstructions. Each call to GET or READ returns one segment. For output records, the application is responsible for breaking the logical record into record segments and for writing the segments out to the device using the PUT or WRITE macros.

Logical Record Interface

CMS simulates the logical record interface (LRI) in QSAM Locate and QSAM Update modes. (CMS also supports the extended logical record interface (XLRI) for longer records, up to 65535 bytes in length.) Under the logical record interface, CMS assembles record segments into complete logical records on input and disassembles logical records into multiple segments on output. By using this interface, an application only has to deal with logical records; it does not have to worry about how the records are segmented or how they are recorded on the physical device.

You can specify the logical record interface using the BUILDRCD or DCB macro. In CMS, when the logical record interface is in effect, all records (spanned or nonspanned) are presented to the application programs only in the record area.

For files with the record formats of VS and VBS, the logical record interface is supported for QSAM Locate and QSAM Update modes. For files with the record formats of DS and DBS, the logical record interface is supported for QSAM Locate mode. If other QSAM modes are used with VS, VBS, DS, or DBS records, the results are unpredictable.

Specifying LRI with the BUILDRCD Macro

You can use the BUILDRCD macro to request the logical record interface (LRI). For example:

```
          .
          .
          .
          BUILDRCD BUFFPOOL,4,100,RECAREA,1032
          OPEN     SPANDCB,OUTPUT
          .
          .
          CLOSE    SPANDCB,OUTPUT
          .
SPANDCB   DCB      DSORG=PS,DDNAME=SPANNER,MACRF=(GL),
                   LRECL=1000,
                   RECFM=VS,BLKSIZE=100,BUFNO=4,
                   BUFCB=BUFFPOOL
          .
          .
          DS       0F       Fullword alignment
BUFFPOOL  DS       XL412    Buffer pool
          DS       0D       Doubleword alignment
RECAREA   DS       XL1032   Record area
```

Figure 61. Using the BUILDRCD Macro and the Logical Record Interface.

The BUILDRCD macro sets up a buffer pool and a record area in a user-provided storage area. In Figure 61 on page 376, this storage area is defined as BUFFPOOL and RECAREA. QSAM uses the areas set up by BUILDRCD for buffering and for assembling and disassembling logical records.

You request the logical record interface through the BUILDRCD by issuing it before a file is opened (as shown in Figure 61 on page 376) or during processing of the DCB open exit routine.

In Figure 61 on page 376, the record format is defined as variable spanned. If LRI is requested and the record format is not VS, VBS, DS, or DBS the request for LRI is ignored and the file is processed according to the specified record format.

Specifying LRI with the DCB Macro

You can also specify the logical record interface using the BFTEK=A option on the DCB macro. If the BUILDRCD macro had not been used in Figure 61 on page 376 then the following DCB macro could have been used to specify LRI.

```
SPANDCB   DCB      DSORG=PS,DDNAME=SPANNER,MACRF=(GL),
                   LRECL=1000,
                   BLKSIZE=100,RECFM=VS,BFTEK=A
```

OS simulation DCB macro processing accepts LRI or XLRI conventions for QSAM variable spanned records, up to 65535 bytes in length. For the logical record length (LRECL) on the DCB macro, CMS supports an absolute value of 9-65535 bytes or 1-64K. Specifying 0K or X indicates the default (maximum) size.

Note: Specifying 64K actually yields a size of 64K-1, or 65535, the maximum LRECL supported.

Determining Record Area Size

If the record area had not been previously allocated using BUILDRCD, then the OPEN macro will allocate a record area for the application to use (see Table 39 on page 377 for the default values). In this case, the associated CLOSE macro will release the record area. If a record area already exists, OPEN will use it instead of allocating a new one.

If the record area is not present and BFTEK=A is specified when a file is opened, CMS defines a record area. The record area size is the logical record length (LRECL) specified in the DCB or by FILEDEF plus 32 bytes. If LRECL is not provided in the DCB or by FILEDEF, the default values for the record area length will be set according to the following table.

Table 39. Default Values for the Record Area

Device	Default Record Area Length
CMS Disk	• 32756+32, if the DCB indicates LRI processing • 65535+32, if the DCB indicates XLRI processing (LRECL= 0K or X)
Tape	• If the file exists, and a tape label is available, the LRECL specified in the data set label for the file is used. • If the file does not exist or no tape label is read: – 32756+32, if the DCB indicates LRI processing – 65535+32, if the DCB indicates XLRI processing (LRECL= 0K or X)
OS Disk	• If the file exists, the LRECL specified in the data set label for the file is used. • If the file does not exist: – 32756+32, if the DCB indicates LRI processing – 65535+32, if the DCB indicates XLRI processing (LRECL= 0K or X)
Printer	• 32756+32, if the DCB indicates LRI processing • 65535+32, if the DCB indicates XLRI processing (LRECL= 0K or X)
Console	• 32756+32, if the DCB indicates LRI processing • 65535+32, if the DCB indicates XLRI processing (LRECL= 0K or X)
Reader	• 32756+32, if the DCB indicates LRI processing • 65535+32, if the DCB indicates XLRI processing (LRECL= 0K or X)

Table 39. Default Values for the Record Area (continued)

Device	Default Record Area Length
Punch	• 32756+32, if the DCB indicates LRI processing • 65535+32, if the DCB indicates XLRI processing (LRECL= 0K or X)
Dummy	• 32756+32, if the DCB indicates LRI processing • 65535+32, if the DCB indicates XLRI processing (LRECL= 0K or X)

These default record area sizes are designed to hold the largest records allowed by LRI (or XLRI) processing. However, allowing the record area size to default can degrade the performance of the application because large amounts of storage are used. If the largest records are less than 32K long (or longer than 32K but less than 64K), you should specify the LRECL in the DCB macro or FILEDEF command as the length of the largest record the application will encounter. This will cause the record area length to be set to the minimum required size. Alternatively, if you use the BUILDRCD macro, you can specify the record area length on the macro to be the size of the largest record plus 32 bytes.

Opening Data Sets

You can open a data file using the OS simulated OPEN macro. You can use the OPEN macro parameters to identify the method of processing and volume position on an end of volume condition. When the OPEN macro instruction is executed, the OPEN routine:

- Completes the data control block. (For more information on how the DCB is built, see "Filling in the DCB information" on page 382 or the *DFSMS Macro Instruction for Data Sets*.)
- Loads all necessary access method routines not already in virtual storage.
- Initializes tape data sets by reading or writing labels and control information.
- Builds the necessary system control blocks and obtains the required buffers.

See Table 28 on page 320 for a list of all the OPEN parameters supported through OS simulation.

Specifying an Input Data File

To specify that a data file be used for input, use the following OPEN parameters:

- For QSAM processing, use INPUT and RDBACK

  Note: RDBACK is only valid for tape files. If RDBACK is specified for a non-tape file, the OPEN parameter will default to INPUT.
- For BSAM processing, use OUTIN, INPUT, or INOUT.

  Note: INOUT cannot be used with AL or AUL tapes.

Reading tape files backward

You can also specify that an input file on magnetic tape be processed backward by using the RDBACK parameter. The RDBACK option is supported for:

- QSAM tape files
- Records formats of F, FB, and U
- All types of label processing
- Single volume magnetic tape files.

The following table describes the tape positioning for files that are opened and closed for RDBACK.

Table 40. Positioning of Files Opened and Closed with RDBACK

Macro	Specifies LEAVE on the OPEN or CLOSE Macro	FILEDEF TAPn Options	Tape positioning
OPEN	Yes	SL|SUL|AL|AUL|NSL option	Assumed position is immediately after the tape mark that ends the trailer label group.
OPEN	Yes	LABOFF|BLP|no label processing option	Assumed position is immediately after the tape mark that ends the data.
OPEN	Yes	No label processing option	The tape is rewound.
OPEN	No[15]	All except LABOFF	The tape is rewound.
OPEN	No[15]	LABOFF	Assumed position is immediately after the tape mark that ends the data.
CLOSE	Yes	SL|SUL|AL|AUL|NSL option	The tape is left immediately before the header label group.
CLOSE	Yes	LABOFF, BLP, NL, no label processing option	The tape is left immediately before the first data block of the file.
CLOSE	No[16]		The tape is always rewound.

For more information on the RDBACK option, see the *DFSMS Macro Instruction for Data Sets*.

Specifying an Output Data File

To specify that a data file be used for output, use the following OPEN parameters,

- For QSAM processing, use OUTPUT and UPDAT

  Note: UPDAT is valid only for DASD files. UPDAT is invalid for a file found on a read-only extension.
- For BSAM processing, use OUTIN, INOUT, or EXTEND.

Note: If the processing method is omitted from the OPEN macroinstruction, INPUT is assumed.

Adding records to the end of an existing file

To specify that new records be added to the end of a file, use the EXTEND parameter of OPEN. The EXTEND parameter prepares a tape or DASD file for output and specifies that new records be added to the end of the file.

EXTEND is valid only for QSAM and BSAM files. If it is specified for any other access method or devices other than DASD and tape, it will be treated as though the OUTPUT parameter had been specified.

[15] If neither LEAVE nor any other positioning option is specified on the OPEN macro, then positioning depends on the FILEDEF options. If you specify the LEAVE option on the FILEDEF command, positioning is as if you specified OPEN LEAVE.

[16] If neither LEAVE nor any other positioning option is specified on the CLOSE macro, then the positioning option is taken from the FILEDEF DISP PASS or KEEP parameter, if any was specified. CLOSE with no options and FILEDEF with the DISP and PASS options behaves like CLOSE LEAVE.

You can also specify the DISP MOD parameter on OPEN or the FILEDEF command to indicate that records be added to the end of a file.

DISP MOD, EXTEND, and INOUT cannot be used with AL or AUL tapes.

For more information on OPEN EXTEND or OPEN with DISP MOD, see the *DFSMS Macro Instruction for Data Sets*. For more information on the DISP MOD option of FILEDEF, see the *z/VM: CMS Commands and Utilities Reference*.

Determination of BLKSIZE and LRECL on CMS DASD

The CMS file system control blocks do not allow for recording both a record length and a block size; only the record length of the file is saved. Generally, the CMS record length will be the OS simulation blocksize. The LRECL and BLKSIZE values used by OS simulation are determined according to the file that is opened.

When a file is opened and either the blocksize or the logical record length is specified, the remaining values are determined according to Table 41 on page 380. These values may be specified either in the DCB or in the FILEDEF command.

Note: If a field is specified in both the DCB and the FILEDEF command, the value from the DCB macro will be used.

If the open intent is not OUTPUT (and thus not replacing the file) and the file already exists on a minidisk or SFS directory and the file is not a partitioned data set, then the block size will be set to the record length of the actual CMS file, regardless of how it was previously specified. For this case, the LRECL can be determined according to Table 41 on page 380.

Table 41. Determining BLKSIZE and LRECL on CMS DASD when either LRECL or BLKSIZE, or both, are specified

BLKSIZE	**LRECL**	**Results**
Specified	Not specified	When simple RECFM values are used: • For RECFM=F: – LRECL=BLKSIZE • For RECFM=V|D|U: – LRECL=BLKSIZE —4 for true OS file types – LRECL=BLKSIZE for CMS file types
Specified	Not specified	When compound RECFM values are used (OS only): • For RECFM=FB|FBS: – LRECL=BLKSIZE • For RECFM=VB|VS|VBS|DB: – LRECL=BLKSIZE —4 • Under LRI or XLRI for RECFM=VS|VBS|DS|DBS: – LRECL=length of record area —32
Not specified	Specified	When simple RECFM values are used: • For RECFM=F: – BLKSIZE=LRECL • For RECFM=V|D|U: – BLKSIZE=LRECL+4 for true OS file types – BLKSIZE=LRECL for CMS file types

Table 41. Determining BLKSIZE and LRECL on CMS DASD when either LRECL or BLKSIZE, or both, are specified (continued)

BLKSIZE	LRECL	Results
Not specified	Specified	When compound RECFM values are used (OS only): • For RECFM=FB|FS|FBS: – BLKSIZE=LRECL • For RECFM=VB|VS|VBS: – BLKSIZE=LRECL+4
Specified	Specified	The specified values are used.

If neither LRECL nor BLKSIZE is specified and, (1) the file is opened for OUTPUT or OUTIN without DISP MOD (meaning that the file will be replaced), or (2) the file being opened does not exist, then message DMSSOP036E will be issued with error code 4. This is because CMS cannot determine what LRECL and BLKSIZE to use. Otherwise, the rules in Table 42 on page 381 apply when neither LRECL nor BLKSIZE is specified.

Table 42. Determining BLKSIZE and LRECL on CMS DASD when neither LRECL nor BLKSIZE is specified.

BLKSIZE	LRECL	Results
Not specified	Not specified	• For RECFM=F|FB|FS|FBS: – BLKSIZE=CMS record length – LRECL=CMS record length • For RECFM=V|VB|VS|VBS|D|DB: – BLKSIZE=CMS record length – LRECL=CMS record length –4 for true OS file types – LRECL=CMS record length for CMS file types • Under the LRI for RECFM=VS|VBS|DS|DBS: – BLKSIZE=CMS file record length – LRECL=length of record area –32

Open Processing

When the OPEN macro instruction is executed, the CMS simulation of the OS OPEN routine initializes the data control block (DCB). After the OPEN macro instruction has been executed, the DCBOFOPN flag in the DCBOFLGS field (bit 3) in the DCB is set if the DCB has been opened successfully, but is not set if the DCB has not been opened successfully.

Unlike OS/MVS, CMS will allow you to perform an open for input on a nonexistent CMS data set. When this occurs, the DCBOFOPN bit in DCBOFLGS is set. The program will find out that the file does not exist only when it does the first I/O on the file and gets an EOF indication. OS simulation behaves this way because certain program products open files which they do not need, which could be empty files in MVS. Because CMS does not really support empty files, OS simulation imitates MVS empty files by allowing a nonexistent file to be opened for input.

An application program using OS simulation services may not open a file for output on a R/O disk. The open bit will not be set, and the DMSSOP036 message will be issued with error code 11.

You should always check the DCBOFOPN bit after an OPEN macro to ensure that the OPEN macro executed successfully. If you do not check this flag and the OPEN macro did not execute successfully, attempts to use the DCB for other OS Simulated Access Method Macros (GET, PUT, and so forth) will lead to unpredictable results. Because OPEN processing sets up the I/O routine address in the DCB, if the

OPEN fails and the application tries to issue a READ, WRITE, GET, or PUT macro, it will branch to an invalid address. The example below shows how you can test whether an OPEN macro completed successfully:

```
          LA    R4,DCBOUT           Point to the DCB
          OPEN  ((R4),OUTPUT)       Open the DCB
          USING IHADCB,R4           Addressability to
                                    the DCB
          TM    DCBOFLGS,DCBOFOPN   Did the DCB OPEN
                                    successfully?
          BZ    OPENERR             No. Go handle OPEN
                                    error
          DROP  R4
           .
           .                 Process a successful OPEN
           .
OPENERR   DS    0H                  Handle OPEN error
          WTO   'OPEN macro failed for OUTFILE'
           .
           .
           .
DCBOUT    DCB   DDNAME=OUTFILE,DSORG=PS,…
           .
           .
           .
          DCBD  DEVD=DA,DSORG=PS     Mapping of the DCB
          END
```

Filling in the DCB information

When the OPEN macro completes successfully, the DCB fields are filled in with information from the DCB macro instruction, the information specified on the FILEDEF command, or, if the data set already exists, the data set label. However, if more than one source specifies information for a particular field, only one source is used.

The precedence of the sources for DCB values is:

1. The values specified on the DCB macro instruction in your program
2. Information in the fields you specified on the FILEDEF command
3. The data set label if the data set already exists.

Any field not set in the DCB macro will be filled in from the FILEDEF command, if it was specified there. Any field not filled in from the DCB macro or FILEDEF command can be filled in from the data set label.

Data set label information from an existing CMS file is used only when the OPEN is for input or update; otherwise, the OPEN routine erases the existing file.

You can modify any DCB field either before the data set is opened or through a data control block open exit. The address of a data control block open exit may be specified using the DCB (EXLST) parameter. For more information on exit routines, see "Exit Routines" on page 387. When the data set is closed, the DCB is restored to its original condition. CLOSE processing clears fields that were merged in at OPEN time from the FILEDEF and the data set label.

Note: The results may be unpredictable if two DCBs access the same file at the same time.

Reading Data

CMS users can read data from both OS/MVS simulated data files and OS/MVS data files.

Reading OS/MVS Simulated Data Sets

For files accessed with the basic access method (BSAM, BPAM, or BDAM), use the READ macro for reading data. For files accessed with the queued sequential access method (QSAM) use the GET macro to read data. Both the basic and queued sequential access methods can read a record with a fixed, variable or variable spanned length.

Using the READ macro

For data sets accessed with one of the basic access techniques, use the READ macro to obtain data from auxiliary storage. READ retrieves a data block from an input data set and places it in a designated area of virtual storage (a buffer). READ processes blocks, not records, so unblocking of records is your responsibility. Buffers where the records are placed can be allocated by you or by the operating system and are filled individually each time a READ is issued.

For fixed and variable-length records

Before issuing READ, R1 must be pointing to the first block to be read. You can then call READ and specify the number of bytes in each block to be read. The length of the blocks can be found in the first 2 bytes of the BDW.

READ will get all of the data records contained in the block and place it at the buffer address specified in the area address parameter of READ. It is then the application's responsibility to break the block into logical records. This can be done using the information in the RDW. See “Record descriptor word” on page 352 for more information.

After READ processing is complete, R1 will be pointing to the next block to be read; READ can then be issued again to obtain the next block of data.

Note: For files opened for RDBACK, register 1 will be pointing to the end of the buffer instead of the beginning.

For variable spanned records

Using the READ macro to get variable spanned records is the same as for fixed and variable records except that your program must make sure that all segments of the records are read. Variable spanned records use the SDW instead of the RDW. The SDW contains information on the length of a variable spanned record segment and whether a segment is the first, middle, last, or only part of the spanned record. See “Segment descriptor word” on page 354 for more information.

The READ macro only starts input operations. To ensure that the operation completes successfully, you should issue a CHECK macro to test the data event control block (DECB). Otherwise, you will not get any notification of I/O errors or end-of-file conditions. For more information on the CHECK macro, see “Using the CHECK macro” on page 385.

Using the GET macro

For data sets accessed with the queued sequential access method (QSAM), use the GET macro to obtain data from auxiliary storage. GET obtains a record from an input data set (as opposed to a block) and places it in a designated area of virtual storage (a buffer). Buffers can be allocated by you or by the operating system and are filled each time a GET is issued.

GET provides various modes that can be specified in the DCB macro. These modes include:

- Locate
- Move.

For more information on these modes, see the *DFSMS Macro Instruction for Data Sets*.

For fixed and variable-length records

Before issuing GET, R1 must be pointing to the first record to be read. You can then call GET and specify the number of bytes in each record to be read. The length of the records can be found in the first 2 bytes of the RDW. See “Record descriptor word” on page 352 for more information.

GET will obtain the record and place it at the buffer address specified in the area address parameter of GET. After GET processing is complete, R1 will be pointing to the next record to be read; GET can then be issued again to obtain the next data record.

For variable spanned records under the segment interface

Using the GET macro to read variable spanned records is the same as for fixed and variable records except that your program must make sure that all segments of the record are read. Variable spanned records use SDW instead of RDW. The SDW contains information on the length of a variable spanned record segment and whether a segment is the first, middle, or last part of the entire spanned record. See "Segment descriptor word" on page 354 for more information.

For variable spanned records under the logical record interface

The logical record interface (LRI) can be used in QSAM Locate mode. LRI will assemble record segments into complete logical records on input so you do not have to worry about how the records are segmented or how they are recorded on the physical device.

LRI can be specified on the DCB or BUILDRCD macro. For more information on LRI and an example, see "Logical Record Interface" on page 376.

Reading OS/MVS Data Sets

CMS users can read OS/MVS sequential and partitioned data sets that reside on OS/MVS disks. The CMS MOVEFILE command can be used to manipulate those data sets, and the OS/MVS QSAM, BPAM, and BSAM macros can be executed under CMS to read them.

The following OS/MVS BSAM, BPAM, and QSAM macros can be used with CMS to read OS/MVS data sets and DOS files:

BLDL	ENQ	RDJFCB
BSP	FIND	READ
CHECK	GET	SYNADAF
CLOSE	NOTE	SYNADRLS
DEQ	POINT	WAIT
DEVTYPE	POST	

Note: These macros cannot be used to read DOS files on fixed block architecture (FBA) devices.
CMS supports the following disk formats for the OS/MVS and OS/VS sequential and partitioned access methods:

- Split cylinders
- User labels
- Track overflow
- Alternate tracks.

As in OS, the CMS support of the BSP macro produces a return code of 4 when trying to backspace over a tape mark or when a beginning of an extent is found on an OS/MVS data set. If the data set contains split cylinders, an attempt to backspace within an extent resulting in a cylinder switch also produces a return code of 4. When a data set has been allocated or updated by OS/MVS on an OS/MVS disk, an OS/MVS CLOSE must be issued before CMS can read or move it. Attempting to read an empty data set on an OS disk is not supported and the results will be unpredictable.

Restrictions for Reading OS/MVS Data Sets

The following restrictions apply when you read OS/MVS data sets from OS/MVS disks under CMS:

- Read-password-protected data sets are not read.
- RACF™ password protection is ignored.
- BDAM and ISAM data sets are not read.

- Multivolume data sets are read as single-volume data sets. End-of-volume is treated as end-of-file and there is no end-of-volume switching.
- Keys in data sets with keys are ignored; only the data is read.
- Results may be unpredictable if two DCBs access the same data set at the same time.
- An Indexed VTOC on an OS/MVS disk is read the same as a standard OS/MVS VTOC because there is no special support in CMS for this.

The following restrictions apply when you are reading OS/MVS data sets from tapes under CMS:

- Read-password-protected data sets are read.
- RACF password protection is ignored.
- User labels in user-labeled data sets are bypassed.
- Results may be unpredictable if two DCBs access the same data set at the same time.

Using the CHECK macro

The CHECK macro tests successful completion of BSAM, BDAM, and BPAM I/O. The CHECK macro refers back to the DECB for the prior READ or WRITE operation corresponding to that DECB. If that I/O operation completed successfully, control returns to the next instruction and the program continues normally. However, if the I/O operation was unsuccessful, CHECK processing will pass control to the address specified on the EODAD or SYNAD parameter of the DCB macro. The error handling routine can perform error analysis, record the error and continue, and/or issue its own abend.

When you code BSAM or BPAM I/O, you must code a CHECK macro for each READ or WRITE macro. When you code BDAM I/O, you must code either a CHECK or a WAIT macro for each READ or WRITE macro. If you do not, your program will continue executing, in spite of any I/O errors or end of file condition. This could cause a variety of situations, including abends, bad data, or destroyed files.

When READing a BPAM or BSAM file, if the program issues a CHECK macro following a READ macro which tried to read a record beyond the last record in the file, control will pass to the EODAD address to handle the condition. Normally, the program should CLOSE the file at this point.

The following is an example of using the READ, WRITE, and CHECK macros.

```
        OPEN  INDCB
        OPEN  OUTDCB
        ...
LOOP    EQU   *
        READ  INDECB,SF,INDCB,BUFFER
        ...
        CHECK INDECB
        ...
        WRITE OUTDECB,SF,OUTDCB,BUFFER
        ...
        CHECK OUTDECB
        ...
        B     LOOP
        ...
ERROR   EQU   *
    ...write out error message ...
END     EQU   *
         ...
         ...
        CLOSE INDCB
         ...
         ...
        CLOSE OUTDCB
         ...
         ...
        BR    R14

  INDCB  DCB DDNAME=INDCB,DSORG=PS,MACRF=(R),
       EODAD=END,SYNAD=ERROR
  OUTDCB DCB DDNAME=OUTDCB,DSORG=PS,MACRF=(W),SYNAD=ERROR
  BUFFER DS  CL800
```

Writing OS/MVS Simulated Data Sets

For files accessed with the basic access method (BSAM, BDAM, or BPAM), use the WRITE macro for writing data. For files accessed with the queued sequential access method (QSAM), use the PUT macro for writing data. Both WRITE and PUT can write a record with fixed, variable, or variable spanned lengths.

Using the WRITE Macro

For data sets accessed with one of the basic access techniques, use the WRITE macro to place data into auxiliary storage. Each time WRITE is issued, data blocks from a buffer in virtual storage are put into an output data set. Because WRITE processes blocks, you are responsible for blocking the records before writing them to virtual storage.

When ASCII/EBCDIC translation is specified (using OPTCD Q on FILEDEF or DCB), if you issue multiple WRITE macros instructions on the same record, an error will occur. This is because the first WRITE instruction issued translates the output data in the output buffer into ASCII.

For fixed and variable-length records

Before issuing WRITE, R1 must be pointing to the place where you want to write the block. You must then assemble the records you want to write into blocks and write the BDWs and RDWs for each block and record written. For information on the contents of the BDW and RDW, see “Block descriptor word” on page 352 and “Record descriptor word” on page 352.

After WRITE processing is complete, R1 will be pointing to the address where next block should be written.

For variable spanned records

Using the WRITE macro to place variable spanned records into auxiliary storage is the same as for fixed and variable records except that your program must make sure that all segments of the records are written. You can put the spanned records into one or more blocks and write an SDW for each segment of the record. The SDW contains information on both the length of the segment and whether it is the first, middle, or last part of the whole spanned record. See “Segment descriptor word” on page 354 for more information.

Checking for I/O completion

The WRITE macro only starts output operations. To ensure that the operation completes successfully, you should issue a CHECK macro to test the data event control block (DECB). Otherwise, you will not get any notification of I/O errors or end-of-file conditions. For more information on the CHECK macro, see “Using the CHECK macro” on page 385.

Using the PUT Macro

For data sets accessed with the queued sequential access method (QSAM), use the PUT macro to write data to auxiliary storage. PUT places a record from a buffer in virtual storage into an output data set.

Similar to the GET macro, PUT provides various modes that can be specified in the DCB macro. These modes include:

- Locate
- Move.

For more information on these modes, see the *DFSMS Macro Instruction for Data Sets*.

For fixed and variable-length records

The first PUT you issue will point R1 to the place where you want to write the first record. Then you can issue another PUT to store the RDW and data in the buffer to the output data set. (For information on the contents of the RDW, see “Record descriptor word” on page 352.) The second PUT also moves the pointer to the address where the next data record will be written. All subsequent PUTs will write the data and move the pointer. When you close the file, the CLOSE macro writes the last data record.

For variable spanned records under the segment interface

Using the PUT macro to write variable spanned records is the same as for fixed and variable records except that your program must make sure that all the segments of the record are written. For each segment of a variable spanned record, a SDW must be written. The SDW contains information on both the length of the segment and whether it is the first, middle of last part of the whole spanned record.

Note that CMS does not clear the third and fourth byte of the SDW on QSAM output. Your application must include the segment control code in the third byte and X'00' in the fourth byte.

See “Segment descriptor word” on page 354 for more information on SDWs.

For variable spanned records under the logical record interface

The logical record interface can be used in QSAM Locate mode. LRI disassembles multiple segments and writes them to auxiliary storage.

LRI can be specified on the DCB or BUILDRCD macro. For more information on LRI and an example, see “Logical Record Interface” on page 376.

Closing Data Files

When you have completed your I/O to a DCB, you must close the data file by issuing the OS simulated CLOSE macro. CLOSE terminates processing of a data set and releases it from the DCB. It restores the original status of the DCB. CLOSE also performs any volume positioning you specified (for tapes only).

A FREEPOOL macro should usually follow a CLOSE macro to regain the buffer pool storage space and allow a new buffer pool to be built if the DCB is reopened with different record size attributes.

Exit Routines

The DCB macro can be used to identify the location of a:

- Routine that performs end-of-data procedures
- Routine that supplements the operating system's error recovery routine
- List that contains addresses of special exit routines.

Table Table 43 on page 387 lists the OS/MVS exit routines that CMS can simulate.

Table 43. OS/MVS Exit Routines Simulated in CMS

Exit Routine	When Available	Where Specified
End-of-data-set	When no more sequential records or blocks are available.	EODAD operand
Error analysis	After an uncorrectable input/output error	SYNAD operand
DCB open	When opening a data set	EXLST operand and exit list
User Label Exits	When processing SUL or AUL tapes.	EXLST operand and exit list
Block count	After unequal block count comparison by end-of-volume	EXLST operand and exit list
DCB Abend	When an abend condition occurs in OPEN, CLOSE, or end-of-volume routine.	EXLST operand and exit list

Following is a description of these exit routines. For more details, see the *DFSMS Macro Instruction for Data Sets*.

End-of-Data-Set Exit Routine (EODAD)

The EODAD parameter of the DCB macro specifies the address of your end-of-data-set routine. This routine can perform any final processing on an input data set. EODAD is entered when an FEOV macro is issued or when a CHECK or GET macro is issued and there are no more records or blocks to be retrieved.

You program will abnormally end under either of the following conditions:

- No exit routine is provided.
- A GET macro is issued in the EODAD routine to the DCB that caused this routine to be entered.

When control is passed to the EODAD routine, the registers contain the following information:

Register	Contents
0-1	Reserved
2-13	Contents before execution of CHECK, GET, or FEOV macros
14	Address of the instruction after the last issued GET, CHECK, or FEOV macro
15	Reserved

Synchronous Error Routine Exit (SYNAD)

The SYNAD parameter of the DCB macro specifies the address of an error routine that is to be given control when an input/output error occurs. This routine can be used to analyze exceptional conditions or uncorrectable errors. The block being read or written, can be accepted, skipped, or processing can be terminated.

Exit List (EXLST)

The EXLST parameter of the DCB macro specifies the address or a list that contains the addresses of special processing routines. An exit list must be created if user labels, data control block, end-of-volume, block count, or DCB abend exit are used.

The exit list is built of 4-byte entries that must be aligned on fullword boundaries. Each exit list entry is identified by a code in the high-order byte, and the address of the routine, image, or area is specified in the 3 low-order bytes. Codes and addresses for the exit list entries are shown in Table 44 on page 388.

Table 44. Format and Contents of an Exit List

Entry Type	Hexadecimal Code	Purpose
Inactive entry	00	Ignore the entry; it is not active
Input header label exit	01	Process a user input header
Output header label exit	02	Create a user output header label
Input trailer label exit	03	Process a user input trailer label
Output trailer label exit	04	Create a user output trailer label
Data control block exit	05	Take a data control block exit.
End-of-volume exit	06	Not simulated

Table 44. Format and Contents of an Exit List (continued)

Entry Type	Hexadecimal Code	Purpose
JFCB exit	07	Not simulated
	08-09	Reserved
User totaling area	0A	Not simulated
Block count exit	0B	Take a block-count-unequal exit
Defer input trailer label	0C	Not simulated
Defer nonstandard input trailer label	0D	Not simulated
	0E-0F	Reserved
FCB image	10	Not simulated
DCB abend exit	11	Examine the abend condition and select one of several options.
QSAM parallel input	12	Not simulated
Allocation retrieval list	13	Not simulated
	14	Reserved
JFCBE exit	15	Not simulated
	16	Reserved
OPEN/EOV nonspecific tape volume mount	17	Not simulated
OPEN/EOV volume security/ verification	18	Not simulated
	19-7F	Reserved
Last entry	80	Treat this entry as the last entry in the list. This code can be specified with any of the above but must always be specified with the last entry.

When control is passed to an exit routine, the registers contain the following information:

Register	Contents
0	Variable; see exit routine description
1	The 3 low-order bytes contain the address of the DCB currently being processed or the parameter list of the exit. See the explanation of each exit routine in the *DFSMS Macro Instruction for Data Sets*.
2-13	Contents before execution of the macro.
14	Return address (must not be altered by the exit routine)
15	Address of exit routine entry point.

The conventions for saving and restoring register contents are as follows:

- The exit routine must preserve the contents of register 14. It need not preserve the contents of other registers. The control program restores the contents of register 2 to 13 before returning control to your program.
- The exit routine must not use the save area whose address is in register 13 because this area is used by the control program. If the exit routine calls another routine or issues supervisor or data management macros, it must provide the address of a new save area in register 13.
- The DCBOFUEX bit in the DCBOFLGS is set to zero when a user exit is taken. This restricts the use of certain macros in an exit routine.

Following is a detailed description of how CMS simulates the DCB abend exit. For information on the other exit lists, see the *DFSMS Macro Instruction for Data Sets*.

DCB Abend Exit (DCB EXLST Entry Code X'11')

The DCB abend exit is provided to handle abend conditions that may occur during:

- Open processing
- Close processing
- End of volume processing

for a DCB.

Figure 62 on page 390 shows the parameter list that contains information about the abend condition. The address of this parameter list is passed to your DCB abend exit in register 1. All information in the parameter list is in binary form.

+0	System Completion Code	Return Code	Option Mask
+4	DCB Address		
+8	Reserved		
+12	Reserved		

Figure 62. DCB Abend Exit (X'11') Parameter List

1. The **system completion code** is 2 bytes long and indicates what error occurred. This will be a right-justified, halfword value. A list of the system completion codes appears in "DCB Abend Exit System Completion Code Values" on page 393.
2. The **return code** is 1 byte long and is a more detailed description of the error.
3. The **option mask** is 1 byte long and contains flags on input to your exit routine indicating whether your application can ignore the abend condition or immediately abend.
4. The **DCB address** is 4 bytes long and contains the address of the DCB on which the error occurred.
5. The **reserved** fields are for IBM use. These fields will contain zeros.

You can use the DMSABEXP macro to map the DCB abend exit parameter list. For more information on DMSABEXP, see the *z/VM: CMS Macros and Functions Reference*.

Note: Although CMS retains the term "abend" for that MVS processing option, this implies CMS error handling for the error case; an abend may or may not be part of the error handling.

Option Mask Values

Your application can indicate whether to ignore the abend condition or immediately abend. To do this, your DCB abend exit routine must determine what action is available for the particular abend condition by examining the contents of the option mask byte in the parameter list (see Figure 62 on page 390). The option mask can have one of the values listed below on input to your exit routine. All other values are reserved for IBM use.

- **X'00'** indicates that CMS error processing will occur regardless of what you specify in the parameter list upon return to CMS. This is specified when there is a irrecoverable error such as an out-of-storage condition.
  - If this occurs during open processing, the DCB is restored to the state it was in prior to calling OPEN and control returns to the application with the DCBOFOPN bit (X'10') in DCBOFLGS set to zero.
  - If this occurs during close processing, an abend 001 occurs.
  - If this occurs during end of volume processing, an I/O error is returned which will invoke the application's SYNAD exit.
- **X'04'** indicates that the exit can choose to ignore this error; abends will be avoided. When this value is used, no error messages will be displayed on the screen; all errors will be transparent to the user.

  Note: You cannot always ignore errors. For these cases, normal CMS error processing occurs regardless of what the exit routine puts in the option byte.
  - If this occurs during open processing, CMS stops processing the DCB and restores the DCB to the state it was in prior to calling OPEN. Control is returned to the application with the DCBOFOPN bit in DCBOFLGS set to zero.
  - If this occurs during close processing, processing stops and the DCB may be partially restored. If you use the DCB at this point, the results are unpredictable.
  - If this occurs during end of volume processing, processing stops. The tape positioning is unpredictable. The application is free to continue processing but any further attempts to use this file (other than closing it, if the error was not an OPEN error) will receive unpredictable results.

Before your DCB abend exit returns control to CMS, your routine places a code in the option mask field to tell CMS what action to take.

- **'0'** indicates that normal CMS error processing for this error will take place.
- **'4'** ignores the abend error. The ignore available bit (X'04') must have been set to '1' on entry to the exit to use this return option.
- Any other option mask value is treated as a '0' and normal CMS error processing will take place.

For more information on the DCB abend exit, see the *DFSMS Macro Instruction for Data Sets*.

Example Using the DCB Abend Exit (X'11')

Following is sample code using the DCB abend exit. It also shows how to use the DMSABEXP macro to map the parameter list that is passed to the DCB abend exit.

Sample Code Using the DCB Abend Exit

```
SAMPABEX CSECT ,
         USING *,R12             Get addressability to our
                                 CSECT
         LR    R12,R15           R12 is our base register
         SPACE
         OPEN  (DCB1,OUTPUT)     OPEN the DCB for OUTPUT.
                                 If an error occurs,
                                 MYABEXT will get control
         .
         .
         .
         BR    R14
         EJECT
*----------------------------------------------------------*
*                                                          *
* The following DCB defines a problem program exit list,   *
* EXITLIST. The exit list indicates that the user written  *
* DCB Abend Exit will be called to examine an abend        *
* condition.                                               *
*                                                          *
*----------------------------------------------------------*
```

```
DCB1     DCB  DSORG=PS,DDNAME=DD1,MACRF=(PL),EXLST=EXITLIST
         SPACE
EXITLIST DS   0F                 DCB Exit List
         DC   XL1'91'            X'11' is Abend Exit
                                 code + X'80' to
                                 indicate that this is
                                 last EXLST entry
         DC   AL3(MYABEXT)       Address of DCB Abend Exit
         SPACE

*----------------------------------------------------------*
*                                                          *
* This is the DCB Abend Exit.  It receives control from    *
* OPEN processing when an error occurs.  For certain       *
* FSOPEN/FSCLOSE errors, tape I/O errors, and general      *
* OPEN errors, this exit will determine the exact cause    *
* of the error, issue a message, and return, telling CMS   *
* to ignore the error.  For all other cases, control       *
* returns with an abend indication, telling CMS to perform *
* its normal error processing.                             *
*                                                          *
*  Upon entry,                                             *
*  R1    = Address of the DCB Abend Exit parameter list    *
*  R2-13 = Contents before the OPEN macro was issued       *
*  R14   = Return address in CMS                           *
*  R15   = Address of MYABEXT                              *
*                                                          *
*----------------------------------------------------------*
MYABEXT  DS    0H                DCB Abend Exit Routine
         USING ABENDEXP,R1       Get addressability to
                                 input plist
         CLI   ABEXOPT,ABEIGNOR  Can we ignore this error?
         BZ    NOIGNORE          No, just return to CMS
                                 Yes, let's see what
                                 the error is
         SPACE
         CLC   ABEXSCC,=H'24'    FSOPEN/FSCLOSE error?
         BE    FSOPNERR          Yes, go handle error
         CLC   ABEXSCC,=H'39'    Tape I/O error?
         BE    TAPIOERR          Yes, go handle error
         CLC   ABEXSCC,=H'36'    General OPEN error?
         BNE   NOIGNORE          No, don't handle the error
*                                Yes, Diagnose OPEN errors
         SPACE
*
*  Diagnose OS OPEN errors
*
         CLI   ABEXRC,RC04       Missing LRECL, BLKSIZE,
                                 or BUFL?
         BNE   …                 No, continue
         LA    R5,MSGnnnn        Yes, get message number
         L     R2,ABEXDCB        Get the DCB address
         USING IHADCB,R2         Get DCB addressability
         LA    R6,LRECLSUB       Get LRECL substitution
         CLC   DCBLRECL,=H'0'    Is LRECL missing?
         BE    IGNORE            Yes, go issue message
         LA    R6,BLKSZSUB       Get BLKSIZE substitution
         CLC   DCBBLKSZ,=H'0'    Is BLKSIZE missing?
         BE    IGNORE            Yes, go issue message
         LA    R6,BUFLSUB        No, must be BUFL;
                                 get substitution
         B     IGNORE            Go issue message
         .
         .
*
*  Diagnose FSOPEN/FSCLOSE errors
*
FSOPNERR DS    0H                Diagnose FSOPEN/FSCLOSE
                                 errors…
         CLI   ABEXRC,RC31       Rollback on SFS file?
         .
         .
*

*  Diagnose Tape I/O errors
*
TAPIOERR DS    0H                Diagnose Tape I/O
                                 errors…
         CLI   ABEXRC,RC05       Tape not attached?
         .
         .
*
```

```
*  Issue diagnostic message and tell CMS to ignore the
*  error.  When control comes here, R7 contains the message
*  number and R6 contains the substitution address.
*
IGNORE   DS    0H                Issue diagnostic message
         APPLMSG NUM=(7),SUB=(CHAR,((6),8))
         MVI   ABEXOPT,ABEIGNOR  Set ignore indicator -
                                 CMS should just move on to
                                 the next DCB
         B     EXITLEAV          Go return to CMS
         SPACE 2
*
*  Tell CMS to proceed with its own error handling
*
NOIGNORE DS    0H                We can't or won't ignore
                                 this error
         MVI   ABEXOPT,X'00'     Tell CMS to do it's
                                 error processing
EXITLEAV DS    0H                Branch address
         BR    R14               Return to CMS
         EJECT
*
*  Pertinent Declarations
*
BLKSZSUB DC   CL8'BLKSIZE'       'BLKSIZE' message
                                  substitution
BUFLSUB  DC   CL8'BUFL'          'BUFL' message substitution
LRECLSUB DC   CL8'LRECL'         'LRECL' message substitution
         .
         .

         DMSABEXP                Abend exit parameter
                                 list mapping
         REGEQU                  Register equates
```

DCB Abend Exit System Completion Code Values

The values returned in the System Completion Code field are shown below. The return code field will contain zero unless otherwise specified.

OPEN Error Codes

24
An FSOPEN or FSCLOSE failed and the return code is put in the Return Code field.

25
Virtual storage capacity exceeded.

28
LABELDEF information for the DCB is missing.

32
An error was detected in an IBM or ANSI label or a nonzero return code was returned from the volume switching routine.

36
OPEN processing encountered an error and the error code was put in the Return Code field. The possible error codes are the same as those returned by DMSOP036E.

39
An I/O error was encountered during tape label processing. The Return Code field will contain the I/O error code. These codes are all of the tape input and output error codes from message DMSxxx120S.

40
Open tried to process a library, but no GLOBAL command was in effect. OS simulation processing on MACLIBs, DOSLIBs, TXTLIBs, and LOADLIBs requires that a GLOBAL command be issued to define the libraries.

CLOSE Error Codes

1
An I/O error occurred when close processing issued the final QSAM PUT for any residual data in the application's output buffers. The Return Code field will contain the I/O error code. These codes can be one of the error code from message DMSxxx120S. Note that since the DMSxxx120S error codes are device dependent, the DCB Abend Exit must be aware of the DCB's device type.

24
An FSCLOSE or FSOPEN error occurred. The FSCLOSE or FSOPEN return code will be placed in the Return Code field.

25
Virtual storage capacity exceeded.

28
LABELDEF information for the DCB is missing.

32
An error was detected in an IBM or ANSI label or a nonzero return code was returned from the volume switching routine.

39
An I/O error was encountered during tape label processing. The Return Code field will contain the I/O error code. These codes are all of the tape input and output error codes from message DMSxxx120S.

EOV Error Codes

25
Virtual storage capacity exceeded.

28
LABELDEF or FILEDEF information for the DCB is missing.

32
An error was detected in an IBM or ANSI label or a nonzero return code was returned from the volume switching routine.

39
An I/O error was encountered during tape label processing. The Return Code field will contain the I/O error code. These codes are all of the tape input and output error codes from message DMSxxx120S.

End-of-Volume Processing

The access methods pass control to the data management end-of-volume routine when an end of volume condition occurs. An end of volume condition can be forced using the FEOV macro or can be detected when one of the following is encountered:

- Tapemark on an input tape volume
- Filemark on an input direct access volume
- End-of-data indicator on an input device other than tape or direct access volume
- End-of-tape indicator on output tape.

If multiple volume data sets are specified in your FILEDEF or LABELDEF command, automatic volume switching can be accomplished by the DMSTVS routine. See "OS/MVS Tape Volume Switching" on page 397 for more information.

Forced End-of-Volume Support

Use the FEOV macro to force an end of volume condition on a file. FEOV signals CMS to assume an end of volume condition for the specified QSAM or BSAM DCB and causes end of volume processing to take place before the physical end of the volume.

Forced end of volume processing is the same as the physical end of volume processing except that the FEOV macro allows tape positioning using the REWIND and LEAVE options.

Forced end of volume processing will, by default, rewind and unload a tape. You can use the REWIND and LEAVE options to position the tape at the beginning of the tape or the end of the file being forced to EOV.

Positioning Tapes with the REWIND and LEAVE Options

REWIND requests that the tape volume be positioned at the beginning of a volume. LEAVE requests that the tape volume be positioned at the logical end of the file for a volume.

In order for the REWIND option or LEAVE to execute correctly, one of the following must be true:

- Only one tape is being processed (it is not multivolume).
- The last tape of a multivolume file is being processed.
- An alternate tape drive has been defined for the multivolume file using the ALT option on FILEDEF.

If none of these conditions are true then the tape is rewound and unloaded, regardless of the positioning option specified.

For tapes that are processed forward when LEAVE is used:

- If the tape is the last or only volume of the file, the logical end of file is the physical end of the data: position **3** in Figure 63 on page 395.
- If the tape is not the last volume or not the only volume of the file, the logical end of file is the end of volume label group: position **4** in Figure 63 on page 395.
- If the tape is unlabeled, the logical end of file is the physical end of the data: position **3** in Figure 63 on page 395.

For tapes that are processed backward when LEAVE is used:

- CMS always assumes a single volume file. Therefore, the logical end of file is the physical beginning of the data: position **2** in Figure 63 on page 395. This is true for unlabeled tapes as well.

Following is an example of the basic tape layout for an ANSI and IBM standard label.

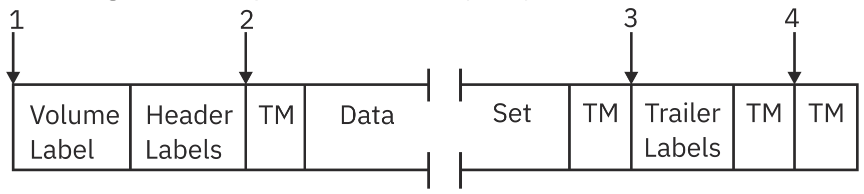


Figure 63. FEOV LEAVE Positioning for ANSI and IBM Standard Labels

If neither REWIND nor LEAVE is specified, the current volume is positioned as follows:

- If this is not the last volume, the tape is rewound and unloaded.
- If this is the last volume and the tape is being processed forward, the tape is positioned after the tape mark that delimits the end of file, **3** in Figure 63 on page 395.
- If this tape is being processed backward, the tape is positioned before the tape mark that delimits the beginning of the file, **2** in Figure 63 on page 395.

Processing Files for Forced End of Volume

The FEOV macroinstruction can be used with magnetic tape files, DASD files, and unit record and terminal files.

For magnetic tape files:

- The REWIND and LEAVE options can be used for volume positioning and are described in the previous section.

- FEOV end-of-volume processing is the same as processing for a physical end-of-volume condition except for the volume positioning options of REWIND and LEAVE. For physical end of volume the tape is always rewound and unloaded.
- EOV output labels are created according to the access method and label processing specification. Also, input labels are checked to ensure the correct volume was mounted.
- The standard OS exit routines that CMS supports are given control as specified in the DCB exit list[17].
- The first GET following an FEOV request for a QSAM input file will read the first block from the new volume and the input request will be satisfied with a record from that block. Any remaining records in the previous block from the last volume will be ignored.
- If the FEOV macro is issued for the last volume of a file, the EODAD (end-of-data-set) user routine is given control. If the FEOV REWIND option is specified, the tape is rewound to the load point, **1** in Figure 63 on page 395. Otherwise, FEOV positions the tape immediately after the tape mark that precedes the EOF label group, **3** in Figure 63 on page 395.
- If LEAVE is specified on the FILEDEF command and the following sequence of macros are issued:
  - FEOV REWIND
  - CLOSE LEAVE
  - OPEN

  errors may occur.

  For this case, it is up to the user to position the tape to the correct file before issuing the OPEN macro.
- If a TEOVEXIT routine is available and the FEOV macro is issued for an SL tape, the exit is given control to handle the end-of-volume condition (in this case, standard CMS multivolume switching routine[18].) For more information on TEOVEXIT routine, see *z/VM: CMS Macros and Functions Reference*.

  The TEOVEXIT routine is ignored for AL tapes.

For DASD files, unit record (readers, punches, printers) files and terminal files:

- If FEOV is issued for an input DCB, the end-of-data-set routine is given control, if it exists.
- If FEOV is issued for an output DCB, it is ignored.

For dummy files, the FEOV macro is ignored.

Error Handling during FEOV Processing

Errors that can occur during FEOV processing are those that occur at the physical end of volume under CMS. These can include end of tape errors, tape label errors, tape I/O errors, and tape volume switching errors. If an error occurs during FEOV processing, the tape remains positioned wherever it was at the time of error and one of the following routines may be called:

- EODAD routine
- DCB abend exit routine
- SYNAD routine (FEOV should not be issued from within SYNAD).

Most errors that occur during input FEOV processing are returned to the application as an EOF condition. When this happens, the EODAD routine will be called, if it is available.

If FEOV is issued for an output tape file under label processing that does not support multivolume tapes, the SYNAD routine will be called after the appropriate tape marks or trailer labels are written. When the SYNAD routine is called, the positioning options are ignored and the tape is left positioned immediately after the final tape mark that was written.

[17] The only DCB exit supported by CMS for an end-of-volume condition is the DCB Abend Exit (exit code X'11'). For end-of-file conditions, CMS supports the user trailer label exit (exit codesX'03' and X'04').

[18] Standard CMS multivolume switching routine refers one of two routines. If the DMSTVI routine has been provided by the user or installation, then it is used. Otherwise, the DMSTVS routine is used.

Input/output errors

Other errors that may occur during FEOV processing are input or output errors. These types of errors will cause control to be passed to either the DCB abend exit or the SYNAD routine, depending on availability. If neither of these routines is available, processing will stop with an abend 001.

If a DCB abend exit routine is available, it will be called with one of the error codes described in "DCB Abend Exit System Completion Code Values" on page 393. If the exit returns an *ignore* for this error, no error will be reflected to the application, but the results of any further attempt to reference the file (aside from closing it) will be unpredictable.

A SYNAD routine will be called if it is specified by the DCB and no DCB abend exit is available. In this case, the SYNADAF macro may be issued to obtain a buffer that contains the text of the error message DMSSCT120S, which includes an error code describing the failure. See the *z/VM: CMS and REXX/VM Messages and Codes* for more information on the error message in z/VM.

Tape volume switching errors

When a program uses QSAM to read a spanned multivolume file, if it issues an FEOV macro and the first segment on the new volume is not the first segment of a record, errors may occur when the next GET macro is issued. Such errors will cause control to be passed to the error analysis exit (SYNAD).

During tape volume switching, if the user fails to mount a new tape (either through a CANCEL command for the mount request or if no response is made to the tape mounting subsystem so that the mount request processing times out), then the CMS system processing will consider the mount failure as an irrecoverable I/O error and force a SYNAD (as specified on the DCB macro) or system ABEND exit to end processing. If the user application contains a SYNAD exit, then that routine will receive control on error termination. Otherwise, the default is to enter a system ABEND exit.

OS/MVS Tape Volume Switching

DMSTVS is a CMS routine that performs tape volume switching operations for OS/MVS multivolume tape support. This routine, called only by OS simulation, prompts the operator to mount a specified tape volume on a specified tape drive.

DMSTVS automatically switches to nucleus key for its processing when it gets control. DMSTVS is called by SVC 202 and must be executed in AMODE 24.

When DMSTVS is called, register 1 points to the address of the tokenized parameter list, whose contents will be as follows:

```
DC    CL8'DMSTVS  '       SVC 202 routine target name
DC    CL8'VOLID   '       Volume id to be mounted or
                          SCRATCH
DC    XL8'01810000'       Virtual tape address
                          (i.e., 181 or
                          any valid CMS tape device
                          number)
DC    CL8'RING    '       Write enable RING or
         'NORING  '       Readonly NORING
DC    CL8'SL      '       IBM Standard labeled tape or
         'AL      '       ANSI Standard labeled tape or
         'NL      '       No labels in use on tape or
         'BLP     '       Bypass label processing or
         'LABOFF  '       Label processing off
DC    CL8'        '       Reserved
DC    CL8'DDNAME  '       FILEDEF that needs this tape
                          volume
DC    CL8'        '       Use device default mount
                          mechanism or
         'LIBSRV  '       RMS will be used to call
                          Library Dataserver
                          (INPUT/OUTPUT field)
DC    8X'FF'              Fence to show end of list
```

The output of DMSTVS depends on the installation. DMSTVS is meant to issue messages to the console to mount tapes. Any non-zero return code is considered an error situation.

The DMSTVS function can be overridden by a nucleus extension with the name DMSTVS. The nucleus extension must follow the same entry and exit conventions as DMSTVS. The exit conditions are:

- Normal: R15 = 0
- Error: R15 does not = 0.

DMSTVS uses the TVSPARMS macro to set the tape volume switching parameters for DMSTVS. The TVSPARMS macro provides time interval information for the volume switching prompt message to the operator. The operator can ask for another reminder to be issued at a preset interval, or cancel the tape mount.

It is possible to change the default values that this macro provides to the DMSTVS module. The macro is used by the native CMS tape switching services to provide:

- The user ID of the person or tape pool operator to receive the tape mounting and error messages from OPERATOR to some other valid ID.
- Default time intervals between tape drive senses
- Default number of message prompts issued before the `Wait time about to expire` and `Wait time expired` messages are issued.

The macro field names and associated default values are as follows:

```
TPUSERID DC   CL8'OPERATOR'  USERID to which mount msgs
                             are sent
WAITIME  DC   C'00003000'    Time to wait between drive
                             senses
*                        (the format of WAITIME is HHMMSSTH)
WAITLPCT DC   F'10'          Number of times to SENSE before
                             prompt
ABOUTEXP DC   F'3'           Prompt count when the
                             "ABOUT TO EXPIRE"
*                            message will be given
EXPIRE   DC   F'4'           Prompt count when the
                             "WAIT TIME EXPIRED"
*                             message will be given
READRING DC   CL8'CHECK'     If NORING was requested,
                             check for RING
```

Figure 64. TVSPARMS Field Default values

To change any of these system defaults, a system programmer must:

1. Change this macro in a local macro library
2. Recompile the DMSTVS module
3. Rebuild the CMS system using the local modification or generate DMSTVS as a nucleus extension module.

Please note that DMSTVS sets a CP timer with the OS/MVS STIMER simulation. This paces the prompting messages.

Usage Notes for DMSTVS Tape Volume Switching

- DMSTVS normally operates in an interactive environment with a human tape operator. It issues repetitive prompt messages based on the TVSPARMS macro WAITIME value and prompt counter expirations.
- DMSTVS prompts are issued both to the virtual machine USERID and to the tape operator identified in the TVSPARMS macro TPUSERID value. The default for the tape operator is OPERATOR, but the macro can be changed by the customer and the code reassembled so that mount prompts go to a specific tape operator user ID.
- Prompt messages have a counter that increases if no response is made by the user or the operator before the next time interval expires. Warnings can also be issued if the total wait time interval is about to expire.

- DMSTVS immediate commands that can be issued are EXTEND and CANCEL. The user or the tape operator can choose to EXTEND the current wait time interval or let it expire, based on circumstances. The user or tape operator can also CANCEL a mount request.
- When a wait time EXTEND is issued, or a mount error occurs (such as when a wrong tape VOLID is mounted for a requested tape), then the prompt counter and time intervals are reset to start over again from zero.
- A DMSTVS time interval between prompts is equal to the TVSPARMS macro WAITIME value multiplied by the TVSPARMS macro WAITLPCT value.
- The DMSTVS total wait time equals the prompt time interval multiplied by the TVSPARMS macro EXPIRE count. This value can be changed if the user or the operator enters the EXTEND immediate command to reset the time intervals to start over again.
- A prompt is also issued warning that the wait time is about to expire when the prompt count equals the TVSPARMS macro ABOUTEXP value.
- DMSTVS sets a non-zero return code to its caller when the total wait time expires, or when a tape mount CANCEL is done. Total wait time expires when the prompt counter equals the TVSPARMS macro EXPIRE value.
- DMSTVS can operate in automated mode if an Automated Tape Library is in control of the target tape drive. In this case, DMSTVS issues CSL Mount or Demount commands to the tape library robot via calls to the DFSMS RMS server machine controlling the tape library. However, if the CSL Mount call to the DFSMS RMS machine ends in error, DMSTVS then tries to issue prompts to the human tape operator identified by the TVSPARMS macro TPUSERID value to get the tape mounted.

Passing Information to the DMSTVI Routine

An interface routine, DMSTVI, can be used to give control to a different multivolume switching routine than the one supplied with VM (DMSTVS) or a tape management system. If a DMSTVI routine is present, it is always used first. You can make a DMSTVI routine available to the system as a nucleus extension.

The load of the DMSTVI module is done at FILEDEF time. Use the SYSPARM option to pass information not included on the FILEDEF or LABELDEF command to the DMSTVI routine. The SYSPARM option of FILEDEF can be used to pass additional information to the DMSTVI routine.

If the customer's tape subsystem product is capable, three file attribute fields are passed in the DMSTVI parameter list expansion area which allows the tape subsystem to record the attributes for reference. These recorded attributes can also be supplied by the tape subsystem at OPEN time as default values to fill in a FILEDEF specification. See the expansion of the TVISECT macro in the *z/VM: CMS Macros and Functions Reference*.

Create the interface routine (DMSTVI) by the normal LOAD/GENMOD procedure. Use the RLDSAVE option of the LOAD command when the text deck is loaded to save the relocation information from the text file.

When DMSTVI is called, the general-purpose registers contain the following information:

GPR 1
: = Address of a parameter list defined by the TVISECT DSECT

GPR 14
: = Return address

GPR 15
: = Entry point address

The calling routine saves and restores the register contents.

DMSTVI automatically switches to nucleus key for its processing when it gets control. When DMSTVI gets control, it must check the call function keyword in the register 1 PLIST. The call function keyword identifies the function being processed when DMSTVI is called. DMSTVI should use the information in the PLIST to build a command or to invoke the tape volume switching routine or tape management system.

When DMSTVI is called during FILEDEF processing, only the call function (SYSPARM) and the SYSPARM string address and length field are filled in. The other fields are set to zeros.

Because DMSTVI gets control during OPEN macro processing before any I/O is done, you do not have to mount a tape before OPEN is issued. The interface routine can mount the tape before returning control to OPEN macro processing.

If you specify only the first volid of a multivolume tape and the end of the first volume is reached, DMSTVI gets control with a call function of 'EOV' and a volid of SCRATC. The tape management system can mount the next volume if it knows what tape is currently mounted on the drive and the volid of the next volume in the series.

A 44 character file ID can now be entered with the LABELDEF command. The 44 character file ID is passed to DMSTVI during OPEN, EOV, and CLOSE macro processing. DMSTVI should check the TVISCRAT field in the register 1 PLIST to determine if a tape was requested from the tape management system. DMSTVI checks the TVISCRAT field by giving a file ID.

If the TVISCRAT field contains 'SCRATCH', a scratch tape was requested. If this field contains 'NOSCRATC', a scratch was not requested — 'SCRATC' was put in TVIVOLID as a default. If a file ID is also specified (TVIFILID), a tape containing this file ID was requested. If you want to mount a tape by specifying just the file ID, you should not specify any volid on FILEDEF or LABELDEF (including 'SCRATCH').

If no file ID is specified on the LABELDEF command, the TVIFID field in the register 1 PLIST contains all zeros. The system uses the *ddname* (TVIFILE) as the default.

The TVIVSEQ field (volume sequence number) is only supported to the extent that the *volseq* operand of the LABELDEF command is supported. This means that the TVIVSEQ field will either be 0 or 1. It is not incremented when multiple volumes are used.

When DMSTVI is called for an end of volume condition or when a file is closed, the TVIBLKCT field will contain a count of the I/O to this tape file. Installations can obtain a count of tape I/O for accounting purposes by recording the TVIBLKCT parameter. Note that this is the count for a file on one particular tape volume. For multivolume tapes, the TVIBLKCT must be accumulated each time DMSTVI is called for file close processing to account for the total I/O count on all the volumes on which the tape file exists. This accounting information is complete only if alternate tape processing is avoided, that is, DMSTVI is not called for alternate processing.

The output of DMSTVI depends on the installation. DMSTVI is meant to issue operator messages to mount a tape for the user application. DMSTVI must return to the calling routine when processing is complete. Any non-zero return code is considered an error situation.

Part 6. DOS/VSE, Access Method Services, and VSAM

This part of the document describes the services of the DOS/VSE operating system as they are simulated by CMS. Note that CMS may not simulate the same services that are provided with DOS/VSE.

Part 6, " DOS/VSE, Access Method Services, and VSAM," on page 401 includes the following chapters:

- Chapter 24, "Developing VSE Programs under CMS," on page 403 describes how you can use CMS to develop and execute your VSE programs.
- Chapter 25, "Using Access Method Services and VSAM," on page 445 describes how you can use CMS to create and manipulate VSAM catalogs, data spaces, and files on OS and DOS disks using access method services.

Chapter 24. Developing VSE Programs under CMS

CMS simulates many functions of the Disk Operating System (DOS) VSE so you can use the interactive facilities of z/VM to develop and execute your VSE programs in a VSE virtual machine or in a batch facility virtual machine. CMS simulation of DOS is not necessarily the same as the current support of DOS. However, you can use CMS to create, compile, test, execute, and debug VSE programs written in the Assembler, DOS/VS COBOL, DOS PL/I, DOS/VS RPG-II programming languages.

This chapter discusses the following topics:

- Entering the CMS/DOS environment
- Using DOS files on DOS disks
- Using libraries in CMS/DOS
- VSE assembler language macros supported
- Assembling, link-editing, and executing source programs
- VSE supervisor and I/O macros supported by CMS/DOS
- CMS/DOS user considerations and responsibilities.

Overview of CMS/DOS

CMS/DOS is neither CMS nor is it DOS. It is a composite, and its vocabulary contains both CMS and VSE terms. CMS/DOS performs many of the same functions as DOS. However, under VSE a function is initiated by a control card, while under CMS it is initiated by a command. Many CMS/DOS commands, therefore, have the same names as the VSE control statement that performs the same function. In those cases where the control statement you would use in VSE and the command you use in CMS are different, the differences are explained. In general, whenever a term that is familiar to you as a VSE term is used, it has the same meaning to CMS/DOS, unless otherwise indicated.

CMS/DOS support in z/VM is based on the VSE licensed program. The term DOS, however, continues to be used in a general sense, and in the discussion that follows, DOS refers to the VSE licensed program.

Entering the CMS/DOS Environment

After you have loaded CMS into your virtual machine, you can enter the CMS/DOS environment by issuing:

```
set dos on
```

A typical response from the system would be:

```
DMSSET1101I 100K DOS partition defined at hexadecimal location 020000
Ready;
```

If you want to access a DOS system residence volume during your CMS/DOS terminal session, you should link to and access the disk that contains the DOS SYSRES before you issue the SET command.

For example, if you share the system residence volume with other users and it is in your directory at virtual address 390, you would issue the command:

```
access 390 g
```

then issue the SET command as follows:

```
set dos on g
```

to indicate that the SYSRES is located on your G-disk. If you are going to use the CMS/DOS librarian facilities to access any of the libraries on the system residence volume, you must enter the CMS/DOS environment this way.

If you are using CMS exclusively for DOS applications, you could put the ACCESS and SET DOS ON commands in your PROFILE EXEC.

All of the CP and CMS online debugging and testing facilities (such as the CP TRACE and STORE commands) are supported in the CMS/DOS environment. Also, CP disk error recording and recovery are supported in CMS/DOS.

CMS/DOS can execute programs that use the sequential access method (SAM) and virtual storage access method (VSAM), and CMS/DOS can access VSE libraries. If you are going to use access method services functions in CMS/DOS or execute functions that read or write VSAM data sets, you must use the VSAM option of the SET DOS ON command:

```
set dos on g (vsam
```

When you are using CMS/DOS, you can use your virtual machine just as you would if you were in the CMS environment. In the CMS/DOS environment, CMS supports many VSE facilities, but does not support OS/MVS simulation. For example, the SCRIPT command uses OS/MVS macros and is therefore invalid in the CMS/DOS environment. When you no longer need VSE support under CMS, you issue the SET DOS OFF command and VSE facilities are no longer available.

You have, however, in addition to the CP and CMS commands available, many CMS/DOS commands and CMS commands with special CMS/DOS operands that simulate VSE functions. Except for the DLBL and DOSLIB commands, these commands or operands should only be issued in the CMS/DOS environment.

The CMS/DOS commands and CMS commands with special CMS/DOS operands are summarized in Table 45 on page 404. A detailed description of the commands and the command format are found in the *z/VM: CMS Commands and Utilities Reference*.

Table 45. CMS/DOS Commands and CMS Commands with Special Operands

Command	Operand	Comments
ASSGN		Assigns CMS/DOS system or programmer logical units to a virtual device. Executable only in the CMS/DOS environment.
DLBL		Defines a VSE or VSAM ddname and relates the ddname to a disk file.
DOSLIB		Deletes, compacts, or lists information about the phases in a CMS/DOS phase library.
DOSLKED		Link-edits CMS text file or object modules from a VSE relocatable library, and places them in executable forms in a CMS/DOS phase library. Executable only in the CMS/DOS environment.
DOSPLI (see note "1" on page 405)		Compiles DOS PL/I source programs. Requires installation of the DOS PL/I Compiler, 5736-PL1. Executable only in the CMS/DOS environment.
DSERV		Displays information about VSE core image, relocatable, source statement, and procedure or transient directories. Executable only in the CMS/DOS environment.
ESERV		Displays, updates, punches, or prints edited (E sublibrary) VSE source statement books. Executable only in the CMS/DOS environment.

Table 45. CMS/DOS Commands and CMS Commands with Special Operands (continued)

Command	Operand	Comments
FCOBOL (see note "1" on page 405)		Compiles DOS/VS COBOL source programs. Requires installation of the DOS COBOL Compiler, 5746-CB1. Executable only in the CMS/DOS environment.
FETCH		Fetches a CMS/DOS executable phase. Executable only in the CMS/DOS environment.
GENMOD	OS DOS ALL	Specifies the type of macro support needed to execute a module. The ALL operand is intended for CMS internal use.
GLOBAL	DOSLIB	The GLOBAL command can specify CMS/DOS phase libraries, as well as text and macro libraries.
LISTIO		Display information about the CMS/DOS system and programmer logical units. Executable only in the CMS/DOS environment.
LOADMOD		Checks that a module generated to execute in a specific macro simulation environment (CMS/DOS or CMS) is in the correct environment.
OPTION		Sets compiler options for DOS/VS COBOL and DOS/VS RPG-II. Executable only in the CMS/DOS environment.
PSERV		Copies and displays procedures in the VSE procedure libraries and spools the procedures to the CMS virtual printer and punch. Executable only in the CMS/DOS environment.
QUERY	DLBL DOSLNCNT DOS DOSLIB DOSPART LIBRARY OPTION UPSI	Displays the current data set definitions. Displays the current number of SYSLST lines per page. Executable only in the CMS/DOS environment. Displays the current status (active or not active) of CMS/DOS. Displays the names of all CMS/DOS phase libraries currently being searched for executable phases. Displays the virtual partition size. Executable only in the CMS/DOS environment. Displays the names of all CMS/DOS phase libraries to be searched, in addition to the text and macro libraries. Displays CMS/DOS compiler options. Executable only in the CMS/DOS environment. Displays current setting of CMS/DOS UPSI byte. Executable only in the CMS/DOS environment.
RSERV		Copies and displays modules in a VSE relocatable library. Output can also be directed to the virtual printer or punch. Executable only in the CMS/DOS environment.
SET	DOS DOSLNCNT DOSPART UPSI	Makes the CMS/DOS environment active or not active. Specifies the number of SYSLST lines per page. Executable only in the CMS/DOS environment. Sets the virtual partition size. Executable only in the CMS/DOS environment. Sets the CMS/DOS User Program Switch Indicator (UPSI) byte.
SSERV		Copies or displays books from the VSE source statement library. Output can also be directed to the virtual printer or punch. Executable only in the CMS/DOS environment.

Note:

1. The files used by this command may be CMS minidisk files, shared files in the Shared File System, or a mixture of both. If a TEXT or LISTING file exists before the command is issued, the files are renamed to 'fn CMSUT1' for the TEXT file, and 'fn CMSUT2' for the LISTING file. This is done to preserve file authorities. It is possible for an error to occur during command processing which would leave the

temporary files on the user's disk. The temporary file may need to be renamed by the user back to the original TEXT or LISTING file. File authorities are not preserved if the user specifies a DLBL with a file type other than TEXT or LISTING for the output text or listing files.

DL/I in the CMS/DOS Environment

Batch DL/I programs can be written and tested in the CMS/DOS environment. This includes programs written in assembler, COBOL, and PL/I languages. Not all functions of COBOL and PL/I are supported. For a description of what is supported, see the documentation on the appropriate licensed program.

Database description generation and program specification block generation can also be executed. However, the application control block generation must be submitted to a DOS virtual machine for execution. The database recovery and reorganization utilities must also be executed in a DOS virtual machine. This support provides the ability to:

- Interactively code DL/I control blocks and application programs that contain imbedded DL/I calls.
- Store and maintain macros used to generate DL/I control blocks in a CMS library. Store and maintain programs created under CMS in a CMS library. Production libraries are thus isolated from the test environment.
- Modify and compile programs using the CMS/DOS text manipulation and EXEC facilities.
- Link-edit and execute batch DL/I programs either interactively or in CMSBATCH. Online DL/I application programs requiring access to CICS*/VS must be submitted to a DOS virtual machine for link-editing, cataloging, and execution.

The following restrictions apply:

- All the existing guidelines and restrictions that apply to VSAM data set creation, maintenance, and application program use apply to DL/I data sets.
- The CMS/DOS restriction on writing to sequential files applies to SHSAM and HSAM.
- To assemble a DBD or PSB under CMS/DOS, you must first copy the DBDGEN and PSBGEN macros from the DOS source statement library to a CMS MACLIB.

For more information about using DL/I in the CMS/DOS environment, see *DL/I DOS/VS Data Base Administration*.

Using DOS Files on DOS Disks

You can have DOS disks attached to your virtual machine by a directory entry or you can link to a DOS disk with the LINK command. You can use the ACCESS command to assign a mode letter to the disk:

```
access 155 b
```

and the RELEASE command to release it:

```
release b
```

Except for VSAM disks, you cannot write on DOS disks or update DOS files on them. You can, however, execute programs and CMS/DOS commands that read from these files, and you can use the LISTDS command to display the file IDs of files on a DOS disk.

For example, if you enter:

```
listds b
```

You receive the following response, if the data set exists:

```
FM DATA SET NAME
B  NEW.TEST DATA
B  ONE.TEST ONE
B  TWO.TEST TWO
```

You can also verify the existence of a particular file. For example, if the file-id is NEW.TEST.DATA you can enter:

```
listds new.test.data.b
      — or —
listds new test data b
```

If the file-id of the DOS file you want to verify contains embedded blanks, for example NEW.TEST DATA, then you have to enter the LISTDS commands with a question mark:

```
listds ? b
```

CMS responds:

```
Enter data set name:
```

and you can enter the exact file-id:

```
new.test data
```

If the data set exists, you receive a response:

```
FM DATA SET NAME
B  NEW.TEST DATA
```

Reading DOS Files

Under CMS/DOS, you can execute programs that read DOS sequential (SAM) files; you can also execute programs that read and write VSAM files. You cannot, however, execute programs to read direct (DAM) or indexed sequential (ISAM) DOS files. Complete information on using CMS to access and manipulate VSAM files is described in Chapter 25, “Using Access Method Services and VSAM,” on page 445.

The discussion below lists the restrictions placed on reading SAM files.

CMS cannot read DOS files that:

- Have the input security indicator on.
- Contain more than 16 user labels and data extents. (If the file has user labels, they occupy the first extent. Therefore, the file must contain no more than 15 data extents.) User labels in user-labeled files are ignored.
- Are multivolume files. Multivolume files are read as single-volume files. End of volume is treated as end of file. There is no end-of-volume switching.

CMS does not support duplicate volume labels. You cannot access more than one volume with the same six-character label while you are using CMS/DOS.

Creating CMS Files from DOS Libraries

You can create CMS files from existing DOS files on DOS disks. CMS simulates the DOS librarian functions DSERV, RSERV, SSERV, ESERV, and PSERV with commands of the same names. You can use these CMS/DOS commands to create CMS files from relocatable source statement or procedure libraries located either on the DOS system residence volume or in private libraries. The functions are fully described later in this section.

Copying DOS Files and Tape Data Files

If you want to create CMS files from DOS files that are not cataloged in libraries or from DOS files on tape, you can use the MOVEFILE command. The MOVEFILE command lets you copy a file from one device to another device of the same or a different type. Before issuing the MOVEFILE command, the input and the output files must be described to CMS with the FILEDEF command.

The MOVEFILE and FILEDEF commands are described in Chapter 22, "Developing OS/MVS Programs under CMS," on page 317 and in the *z/VM: CMS Commands and Utilities Reference*. The procedures are the same for copying DOS files as for OS/MVS data sets. You must remember, however, that:

- Because DOS files on DOS disks do not contain BLKSIZE, RECFM, or LRECL options, these options must be specified with the FILEDEF command. Otherwise, default values are assigned. The default values are BLOCKSIZE=32760 and RECFM=U. LRECL is not used for RECFM=U files.
- If a DOS file ID does not follow OS/MVS naming conventions, you must use the DSN ? operand of FILEDEF to enter the DOS file ID. The OS/MVS naming conventions are: 1-byte to 8-byte qualifiers, each qualifier must be separated by a period, and up to 44 characters including periods.

Copying Modules from VSE Library or SYSIN Tapes

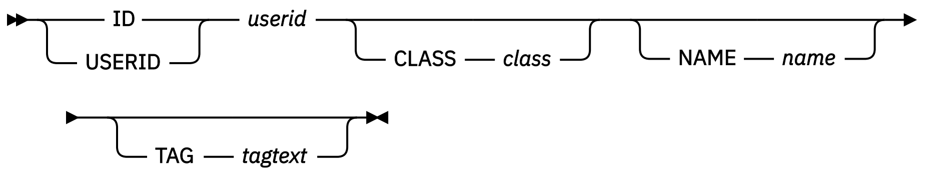
You can create individual CMS files for VSE modules from a VSE library distribution tape or VSE SYSIN tape. Use the VMFDOS command. The VMFDOS command can create a CMS file for each VSE module that exists, and the CMS file name corresponds to the VSE module name. You can restore individual modules, groups of modules, or the entire module set.

For VSE library distribution tapes, the VMFDOS command restores modules from either system or private (relocatable or source statement) libraries. The created CMS files have a file type of TEXT if they are from a relocatable library. They have a file type of MACRO if they are from a source statement library.

Reading in Real Card Decks

If you have DOS files or source programs on cards, you can create CMS files directly by having these cards read into the real system card reader and directed to your virtual machine by preceding the data with a CP ID card.

The format of a CP ID card is:



The ID or USERID keyword must begin in column 1, and each keyword and operand must be separated by at least one blank. *userid* is the user to receive the spool file containing the card deck, *class* and *name* are optional values that can be assigned to the spool file, and *tagtext* is optional data that will be associated with the spool file. If *tagtext* is specified, it must be the last operand on the card.

When the cards appear in your virtual reader, you can read them into a CMS file using any of the usual methods, such as the RECEIVE or READCARD commands.

Using Tapes in CMS/DOS

See "Label Processing in CMS/DOS" on page 164 for a description of CMS tape label processing for CMS/DOS tape files. The support for tape labels is only for files defined by a DTFMT macro. If you do not use this macro, CMS bypasses IBM standard labels on input tapes and writes a tape mark over any existing labels on an output tape. The CMS LABELDEF command is equivalent in CMS/DOS to the VSE TLBL control statement when standard tape label processing is used.

The ASSGN Command

The ASSGN command performs the same function for CMS/DOS as the ASSGN control statement in VSE. The ASSGN command in CMS/DOS assigns a system or programmer logical unit (SYSxxx) to a virtual I/O device. A logical unit is a symbolic name a program may use to refer to a real I/O device without knowing the device address.

If the device is a disk, you can use the DLBL command to establish a real file identification for a symbolic file name in a program. The DLBL command is described under "Using the DLBL Command". As in VSE, you are not allowed to assign the system residence volume with the ASSGN command.

In addition to using the ASSGN command to relate real I/O devices with symbolic units, you must use it in CMS/DOS to:

- Assign SYSIN or SYSIPT for the input source file for a language compiler when you use the DOSPLI or FCOBOL commands.
- Identify the disk, by mode letter, on which a private core image, relocatable, or source statement library resides.
- Assign SYSIN or SYSIPT to the CMS disk on which an ESERV file, containing control statements for the ESERV program, resides.

When you enter the ASSGN command, you must supply the logical unit and the device. For example:

```
assgn sys100 printer
```

assigns the logical unit SYS100 to the printer. When you want to make an assignment to a disk device, you must specify the mode letter where the disk is accessed. You must also access the disk before making an assignment to the disk. The command:

```
assgn sys010 b
```

assigns the logical unit SYS010 to your B-disk.

Assigning System Logical Units

The system logical units you can assign and the valid device types you can assign to the units in CMS/DOS are listed below.

Some VSE system logical units cannot be assigned to a DOS formatted FB-512 device. These units are listed below. An error message is issued and the command terminated if any of the unsupported system logical units are specified in the ASSGN command.

SYSIPT, SYSRDR, SYSIN

You can assign SYSIPT, SYSRDR, and SYSIN to disk (mode), TAPE, or READER. If you make an assignment to SYSIN, both SYSRDR and SYSIPT are also assigned the same device. SYSIPT, SYSRDR, and SYSIN cannot be assigned to a DOS formatted FB-512 device.

SYSLST

You can assign SYSLST, the system logical unit for listings, to disk (mode), PRINTER, or TAPE. An assignment to DOS FB-512 disks is not supported.

SYSLOG

You can assign SYSLOG, terminal or operator output or messages, to PRINTER or TERMINAL. CMS/DOS always assigns SYSLOG to TERMINAL by default, so you never have to make this assignment except when you want to alter it.

SYSPCH

You can assign SYSPCH, punched output (for example, text decks), to PUNCH, disk (mode), or TAPE. An assignment to DOS FB-512 disks is not supported.

SYSCLB, SYSRLB, SYSSLB

You can assign SYSCLB, SYSRLB, and SYSSLB to private core image, relocatable, and source statement libraries, respectively. The only valid assignment for these units is to disk (mode). If you want to reference private libraries with the DOSLKED, DSERV, ESERV, FETCH, SSERV, or RSERV commands, you must assign SYSCLB, SYSRLB, or SYSSLB to the disks where the libraries reside.

Compiler I/O Assignments

The compilers supported by CMS/DOS expect input/output to be assigned to the following devices:

- SYSIN/SYSIPT must be assigned to the device where the input source file resides. Valid device types are reader, tape, or disk.

  You must assign SYSIN/SYSIPT. If it is unassigned at compilation time, an error message is issued and the FCOBOL or DOSPLI command is terminated.
- The user should assign the following logical units to any of the indicated device types:
  - SYSPCH and SYSLST to tape, punch, disk, or IGN

    If SYSPCH or SYSLST are unassigned at compilation time, the FCOBOL or DOSPLI EXEC file directs output to the disk where SYSIN resides if SYSIN is assigned to a read/write CMS disk. Otherwise, output is directed to the CMS read/write disk with the most read/write space.
  - SYSLOG to terminal

    If SYSLOG is unassigned, it is assigned to the terminal.
  - SYS001, SYS002, and SYS006 to disk.

    If SYS001, SYS002, and SYS006 are unassigned, output is directed to the CMS disk with the most read/write space.
  - SYS003-SYS005 to tape or disk.

    If SYS003 through SYS005 are unassigned, output is directed to the CMS disk with the most read/write space.

The maximum number of work files is six for DOS/VS COBOL Compiler (FCOBOL) and two for DOS PL/I Optimizing Compiler (DOSPLI).

All nonpermanent DLBL file definitions are cleared when the DOSPLI or FCOBOL command ends.

Manipulating Device Assignments

You can assign programmer logical units SYS000 through SYS241 with the ASSIGN command. This deviates from VSE where the number of programmer logical units varies according to the number of partitions. Besides assigning I/O devices, the ASSGN command can also negate a previous assignment:

```
assgn syspch ua
```

Also, for a given device, ASSGN can specify that no real I/O operation (NOP) is to be performed during the execution of a program:

```
assgn sys009 ign
```

When you release a disk from your virtual machine, any assignments made to that disk are unassigned.

Listing I/O Assignments

You can find out the current assignments for system and programmer logical units with the LISTIO command, which lists all the system or programmer logical units, even those that are unassigned. To list only currently assigned units, enter:

```
listio a
```

To find out the current assignment of one specific unit, for example SYS100, enter:

```
listio sys100
```

When you use the STAT option, LISTIO lists, for disk devices, whether the disk is read-only or read/write. For example, if you enter

```
listio sys100 (stat
```

you may receive the reply:

```
SYS100 B   R/W
```

This reply indicates that SYS100 is assigned to the B-disk, which is a read/write disk.

With the EXEC option of the LISTIO command, you can create a disk file containing the list of assignments. The name of the file is $LISTIO EXEC. It contains two EXEC numeric variables, &1 and &2, for each unit listed. For example, if you enter the command:

```
listio sys081 (exec
```

the file $LISTIO EXEC may contain the record:

```
&1 &2 SYS081 PRINTER
```

You can cancel all current assignments by leaving the CMS/DOS environment and then re-entering it:

```
set dos off
set dos on
```

Virtual Machine Assignments

When you assign a physical device type to a system or programmer logical unit, CMS relates the device to your virtual machine configuration. You receive an error message if you try to assign a logical unit to a device not in your configuration. For example, if you use the ASSGN command to assign a logical unit to a disk file, you must specify the access mode letter of the disk. If the disk is not accessed, the ASSGN command fails.

For another example, if you issue:

```
assgn syspch punch
```

the punch specified is your own virtual machine card punch. The actual destination of punched output then depends on the spooling characteristics of the punch. If it is spooled to another user or to *, then no real cards are punched; virtual card images are placed in the virtual reader of the destination user ID, which may be another virtual machine or your own.

CMS supports only one reader, one punch, and one printer. You cannot make any assignments for multiple output devices in CMS/DOS. When you make an assignment for a logical unit that has already been assigned, it replaces the current assignment.

The DLBL Command

The DLBL command performs the same functions for CMS/DOS as the DLBL control statement in VSE. Use the DLBL command to supply CMS/DOS with specific file identification information for a disk file that is going to be used for input or output. For any DLBL command you issue, you must previously have issued an ASSGN command for the disk, specifying a system or programmer logical unit. The basic relationship is:

```
assgn SYSxxx mode
dlbl filename mode DSN ? (SYSxxx
```

Both the SYSxxx and the mode values must match on the ASSGN and DLBL commands. The disk where the file resides must be accessed at the specified mode.

Note: File modes R and T cannot be used on this form of the ASSGN command because they are used as abbreviations for reader and terminal. Instead use,

```
assgn SYSxxx diskm
dlbl filename m DSN ? (SYSxxx
```

The file name on the DLBL command line, called a ddname in CMS/DOS, corresponds to the symbolic name for a file in a program. If you want to reference a private DOS library, you must use one of the following file names:

System Logical Unit	File name
SYSCLB	IJSYSCL
SYSRLB	IJSYSRL
SYSSLB	IJSYSSL

Entering File Identifications

When you issue the DLBL command you must identify the file, by file-id (for a VSE file) or by file identifier (for a CMS file). The keywords DSN and CMS indicate whether it is a VSE file or a CMS file, respectively.

If the file is a VSE file residing on a DOS disk, you can enter the DLBL command in one of three ways. For example, for a file named TEST.FILE.INPUT you may enter either:

```
assgn sys101 d
dlbl infile d dsn test.file.input (sys101

     — or —

dlbl infile d dsn test file input (sys101

     — or —

assgn sys101 d dlbl infile d dsn ? (sys101
```

For any VSE file with a file-id that contains embedded blanks, you must use the *DSN ?* form, shown in the third example above. When you enter the DLBL command with the ? operand, you are prompted to enter the DOS file-id:

```
Enter data set name:
```

Then you can enter the DOS file-id. For example,

```
test.file.input
```

When you issue a DLBL command for a CMS file, you enter the file name and file type following the keyword CMS:

```
assgn sys102 a
dlbl outfile a cms new output (sys102
```

In this example, if SYS102 is defined as an output file for a program, the output is written to your CMS A-disk in a file named NEW OUTPUT.

You can, for convenience, use a CMS default file identifier. If you enter:

```
dlbl outfile a cms (sys102
```

then the output file type defaults to the ddname and the file name to FILE. So, this output file is named FILE OUTFILE.

Clearing and Displaying File Definitions

You can clear a DLBL definition for a file by using the CLEAR operand of the DLBL command:

```
dlbl outfile clear
```

To clear all existing definitions, except those entered with the PERM option, you can enter:

```
dlbl * clear
```

This command is issued by the assembler and the language processors when they complete execution. Definitions entered with the PERM option must be individually cleared.

Whenever you use the HX Immediate command to halt the execution of a program, the DLBL definitions in effect are cleared, including those entered with the PERM option.

You can find out what definitions are currently in effect by issuing the DLBL command with no operands:

```
dlbl
```

or you can use the QUERY command with the DLBL operand.

Using DOS Libraries in CMS/DOS

CMS/DOS provides you with the capability of using various types of files from DOS system or private libraries. You can copy, punch, display at the terminal, or print:

- Books from system or private source statement libraries using the SSERV command. Books refer to macros and source programs in a source statement library.
- Relocatable modules from system or private relocatable libraries using the RSERV command.
- Procedures from the system procedure library using the PSERV command.

You can also:

- Copy and de-edit macros from system and private E sublibraries using the ESERV command.
- Access the directories of system or private libraries using the DSERV command.
- Link-edit relocatable modules from system or private relocatable libraries with the DOSLKED command.
- Read core image phases from system or private core image libraries into storage for execution using the FETCH command.

The SSERV Command

If you have cataloged source programs or copy files in the system source statement library and you want to use CMS to modify and test them, you can copy them into CMS files using the SSERV command. For example, suppose you want to copy a book named PROCESS from the A sublibrary on the system residence volume. The DOS system residence is in your virtual machine configuration at virtual address 350, and you have accessed it as your F-disk.

First, to indicate to CMS/DOS that the system residence is on your F-disk, you enter:

```
set dos on f
```

then you can enter the SSERV command, specifying the sublibrary identification and the book name:

```
sserv a process
```

This creates, from the A sublibrary, a file named PROCESS COPY and places it on your A-disk. If the book contained assembler language source statements you would want the file type to be ASSEMBLE, so you may enter:

```
sserv a process assemble
```

If you want to copy a book from a private source statement library, you must first use the ASSGN and DLBL commands to make the library known to CMS/DOS. For example, to obtain a copy file from a private library on a DOS disk accessed as your D-disk, enter:

```
assgn sysslb d
dlbl ijsyssl d dsn ? (sysslb
Enter data set name:
program.test library
```

Now, when you enter the SSERV command:

```
sserv t setup copy
```

the book named SETUP in the T sublibrary of PROGRAM.TEST LIBRARY is copied into a CMS file named SETUP COPY. If SETUP is not found in the private library, then CMS searches the system library, if it is available.

The RSERV Command

In CMS/DOS, to manipulate relocatable modules that have been cataloged either on the system or a private relocatable library you must first copy them into CMS files with the RSERV command. You can link-edit modules directly from DOS relocatable libraries, but if you want to add or modify linkage editor control statements for a module, you must place the control statements in a CMS file.

If you are copying a relocatable module from the system relocatable library, you should make sure that you have indicated the system residence disk when you entered the CMS/DOS environment:

```
set dos on f
```

then you can issue the RSERV command specifying the name of the relocatable module you want to copy:

```
rserv rtna
```

The execution of this command results in the creation of a CMS file named RTNA TEXT on your A-disk.

If you want to copy a relocatable module from a private relocatable library, you must first use the ASSGN and DLBL commands to make the private library known to CMS/DOS:

```
assgn sysrlb d
dlbl ijsysrl d dsn reloc.lib (sysrlb
```

Then, issue the RSERV command for a specific module in that library:

```
rserv testrtna
```

to create the CMS file TESTRTNA TEXT from the module named TESTRTNA. If the module TESTRTNA is not found in RELOC.LIB, CMS searches the system library, if it is available.

The PSERV Command

If you want to copy DOS cataloged procedures into CMS files to use in preparing job streams for a DOS virtual machine, you can use the PSERV command:

```
pserv prepjob
```

This command creates a CMS file on your A-disk. The file is named PREPJOB PROC. To copy a procedure from the procedure library you must have entered the CMS/DOS environment specifying a disk mode for the system residence volume.

You cannot execute DOS/VSE procedures directly from the CMS/DOS environment. However, if you modify a procedure, you can punch it to a virtual machine that is running a DOS system and execute it there.

The ESERV Command

The CMS/DOS ESERV command is actually an exec procedure that calls the VSE ESERV utility program. To use the ESERV program, you first must IPL CMS with a CMSBAM DCSS (shared segment), then create a file with a file type of ESERV that contains the ESERV control statements you want to execute.

For example, if you want to write a de-edited copy of the macro DTFCD onto your A-disk, you might create a file named DTFCD ESERV, with the record:

```
 PUNCH E.DTFCD
```

Just as when you submit ESERV jobs in DOS, column 1 must be blank.

Before executing the ESERV program, you must enter the CMS/DOS environment by specifying the SET DOS ON command using a VSE system residence volume. This is necessary because the ESERV procedure invokes the ESERV program directly from the VSE core image library.

Then, you must assign SYSIN to the device on which the ESERV source file resides, usually your A-disk:

```
assgn sysin a
```

Then you can enter the ESERV command specifying the file name of the ESERV file:

```
eserv dtfcd
```

No other ASSGN commands are required. The CMS/DOS ESERV EXEC makes default assignments for SYSPCH and SYSLST to disk.

To copy and de-edit macros from a private E sublibrary, issue the ASSGN and DLBL commands to identify the library. For example, to identify a source statement library named TEST.MACROS on the DOS disk accessed as the C-disk, enter:

```
assgn sysslb c
dlbl ijsyssl c dsn test.macros (sysslb
```

The SYSLST output is contained in a CMS file with the same file name as the ESERV file and a file type of LISTING. You must examine the LISTING file to see if the ESERV program executed successfully.

The SYSPCH output is contained in a file with the same name as the ESERV file and a file type of MACRO. If you want to punch ESERV output to your virtual card punch, make an assignment of SYSPCH to PUNCH.

When you use the PUNCH or DSPCH ESERV control statements, CATAL.S, END, or /* records may be inserted in the output file. When you use the MACLIB command to add the MACRO file to a CMS macro library, these statements are ignored.

See "Using Macro Libraries" on page 416 for information on creating and manipulating CMS macro libraries.

The DSERV Command

You can use the DSERV command to examine the contents of system or private libraries. If you do not specify any options with it, the DSERV command creates a disk file, named DSERV MAP, on your A-disk. You can use the PRINT or TERM options to specify that the directory list is either to be printed or displayed at your terminal. You can also use the SORT option to create a sorted list.

To examine a system directory, you must have entered the CMS/DOS environment specifying the mode letter of the DOS system residence:

```
set dos on f
```

If you want to examine the directory of a private source statement, core image, or relocatable library you must issue the ASSGN and DLBL commands establishing SYSSLB, SYSCLB, or SYSRLB before using the DSERV command.

For example, to display at your terminal a list of procedures cataloged on the system procedure library, you would issue:

```
dserv pd (sort term
```

If the directory you are examining is for a core image library, you can specify a particular phase name to verify the existence of the phase:

```
dserv cd phase $$bopen (term
```

To list the directory of a private source statement library, you would first issue the ASSGN and DLBL commands:

```
assgn sysslb b
dlbl ijsyssl b dsn test.source (sysslb
```

then enter the DSERV command:

```
dserv sd
```

The CMS file, DSERV MAP A, contains the directory of the private source statement library TEST.SOURCE.

The DOSLKED Command

You can use the DOSLKED command to link-edit relocatable modules from system or private relocatable libraries and to place these modules in a phase library (DOSLIB). CMS searches for a module in a private relocatable library before searching in a system relocatable library.

For more information on using the DOSLKED command, see "Link-Editing Programs in CMS/DOS" on page 420.

DOS Core Image Libraries

You can load core image phases from DOS core image libraries into virtual storage and execute them under CMS/DOS. Because CMS cannot write directly to DOS disks, linkage editor output under CMS/DOS is placed in a special CMS file called a DOSLIB. When you execute the FETCH command in CMS/DOS you can load phases from either system or private DOS core image libraries as well as from CMS DOSLIBs. More information on using the FETCH command is contained under "Executing Programs in CMS/DOS" on page 423.

Using Macro Libraries

DOS macro libraries cannot be accessed directly by the z/VM assembler. If you want to assemble DOS programs in CMS/DOS that use DOS macro or copy files that are on the system or a private macro library, you must first create a CMS macro library (MACLIB) containing the macros you wish to use. Since the process of creating a CMS MACLIB from the DOS system source statement library (E sublibrary) can be very time-consuming, you should check with your installation's system programmer to see if it has already been done and to verify the file name of the macro library, so that you can use it in CMS/DOS.

Note: The DOS PL/I and DOS/VS COBOL compilers executing in CMS/DOS cannot read macro or copy files from CMS MACLIBs. Macros and copy files are obtained instead from a DOS source statement library.

If you want to extract DOS system macros to modify them for your private use, or if you want to use macros from a private library in CMS, you must use the procedure outlined below to create the MACLIB files.

Creating CMS MACLIBs

A CMS macro library has a file type of MACLIB. You can create a MACLIB from files with file types of MACRO or COPY. A MACRO file may contain macro definitions. COPY files contain predefined source statements.

To create a CMS macro library, each macro or copy file you want included in the MACLIB must first be contained in a CMS file with a file type of COPY or MACRO. If you are creating a CMS MACLIB file from a DOS library, you must use the SSERV command to copy a file from any source statement library other than an E sublibrary or use the ESERV command to copy and de-edit a macro from an E sublibrary. The SSERV command uses a default file type of COPY. The ESERV command uses a default file type of MACRO.

The following example shows how to copy macros from various sources and shows how to create and use the CMS MACLIB that contains these macros:

1. Enter the CMS/DOS environment with the DOS system residence on a disk accessed as mode C:

   ```
   set dos on c
   ```

2. Copy the macro book named OPEN from the A sublibrary of the system source statement library:

   ```
   sserv a open
   ```

3. Establish a private source statement library:

   ```
   access 351 d
   assgn sysslb d
   dlbl ijsyssl d dsn ? (sysslb
   test source.lib
   ```

4. Issue the SSERV command for a macro in the M sublibrary of TEST SOURCE.LIB:

   ```
   sserv m releas
   ```

5. Create an ESERV file to copy from the E sublibrary:

   ```
   xedit contrl eserv
   input punch contrl
   file
   ```

6. Execute the ESERV command:

   ```
   assgn sysin a
   eserv contrl
   ```

7. Create a CMS macro library named MYDOSMAC from the files just created, which are named OPEN COPY, RELEAS COPY, and CONTRL MACRO:

   ```
   maclib gen mydosmac open releas contrl
   ```

   See the *z/VM: CMS Application Development Guide* for more details.

8. To use these macros in an assembler language program, you must indicate that this MACLIB is accessible before assembling a source file:

   ```
   global maclib mydosmac
   ```

   See “Identifying Libraries” on page 220 for more details.

Rather than issuing these commands every time you want to copy and create macros, you can put these commands in an exec.

VSE Assembler Language Macros Supported

The programming interfaces defined by the VSE operating system and simulated by CMS are documented below. For definitive information about these interfaces, see the VSE documentation.

A DOS service call made through a DOS/VSE macro while in access-register mode in an XC virtual machine causes an abend. An abend code of X'1CD' indicates this.

Table 46 on page 418 lists the VSE assembler language macros supported by CMS/DOS. You can assemble source programs that contain these macros under CMS/DOS, if you have the macros available in either your own or a shared CMS macro library. The macros whose functions are described in the *Function* column with the term *no-op* are supported for assembly only; when you execute programs that contain these macros, the VSE functions are not performed. To accomplish the macro function you must execute the program on a real VSE system.

Table 46. VSE Macros Supported by CMS

Macro Name	SVC Number	Function
CALL		Pass control to another program
CANCEL	06	Terminate processing
CDLOAD	65	Load a VSAM phase
CHECK		Verify completion of a read or write operation
CLOSE/ CLOSER		Deactivate a data file
CNTRL		Control a physical device
COMRG	33	Return address of background partition communication region
DEQ	41	no-op
DTFxx		Establish file definitions
DUMP		Dump storage and registers and terminate processing
ENQ	42	no-op
EOJ	14	Terminate processing normally
ERET		Provide an error routine
EXCP	00	Execute a channel program
EXIT PC	17	Return from program check routine
EXIT AB	95	Return from abnormal termination routine
EXTRACT	98	Retrieve PUB, storage boundaries, or CPUID information
FCEPGOUT	86	no-op
FETCH	01	Load and pass control to a phase
FETCH	02	Load and pass control to a logical transient
FREE	36	no-op
FREEVIS	62	Release user free storage
GENL		Generate a phase directory list
GET		Access a sequential file
GETFLD/ MODFLD	107	Provide macro interface support for system information retrieval.
GETVCE	99	Return requested device information to output area.
GETVIS	61	Obtain user free storage
GETIME	34	Get the time of day
JDUMP		Dump storage and registers and terminate processing

Table 46. VSE Macros Supported by CMS (continued)

Macro Name	SVC Number	Function
LOAD	04	Load a phase into storage
LOCK/ UNLOCK	110	Resource control
MVCOM	05	Modify bytes in the partition communication region
NOTE		Manage data set access
OPEN/ OPENR		Activate a data file
PAGEIN	87	no-op
PDUMP		Dump storage and registers and continue processing
PFIX	67	no-op
PFREE	68	no-op
POINTR		Position a file for reading
POINTS		Reposition a file to its beginning
POINTW		Position a file for writing
POST	40	Post the event control block
PRTOV		Control printer overflow
PUT		Write to a sequential file
PUTR		Communicate with the system operator
READ		Access a sequential file
RELPAG	85	Simulates the release of pages by setting them to binary zeros
RELSE		Skip to begin reading next block
RETURN		Return control to calling program
RUNMODE	66	Check if program is running real or virtual
SECTVAL	75	Obtain a sector number
SETIME	10/24	no-op
SETPFA	71	no-op
STXIT AB	37	Provide or terminate linkage to abnormal ending routine
STXIT PC	16	Provide or terminate linkage to program check routine
STXIT IT	18	no-op
STXIT OC	20	no-op
SUBSID	105	Retrieve information on supervisor subsystem
TRUNC		Skip to begin writing next block
TTIMER	52	Return a 0 in Register 0 (effectively a no-op)
WAIT	07	Wait for the event completion
WRITE		Write to a sequential file
xxMOD		Create Logical IOCS routine inline

Assembling Source Programs

If you are a DOS assembler language programmer using CMS/DOS, you should be aware that the assembler used is the z/VM assembler, not the DOS assembler. The major difference is that the z/VM assembler, invoked by the ASSEMBLE command, is designed for interactive use. Therefore, when you assemble a program, error messages are displayed at your terminal when compilation is completed, and you do not have to wait for a printed listing to see the results. You can correct your source file and reassemble it immediately. Then you can print the error-free listing.

To specify options to be used during the assembly, you enter them on the ASSEMBLE command line. So, for example, if you do not want the output LISTING file placed on disk, you can direct it to the printer:

```
assemble myfile (print
```

All of the ASSEMBLE command options are listed in the *z/VM: CMS Commands and Utilities Reference*.

When you invoke the ASSEMBLE command specifying a file with a file type of ASSEMBLE, CMS searches all of your accessed disks, using the standard search order, until it locates the file.

When the assembler creates the output LISTING and TEXT files, it writes them onto disk according to the following priorities:

1. If the source file is on a read/write disk, the TEXT and LISTING files are written onto the same disk.
2. If the source file is on a read-only disk that is an extension of a read/write disk, the TEXT and LISTING files are written onto the parent disk.
3. If the source is on any other read-only disk, the TEXT and LISTING files are written onto the A-disk.

In all of the above cases, the file names assigned to the TEXT and LISTING files are the same as the file name of the input file.

The output files used by the assembler are defined with FILEDEF commands issued by CMS when it calls the assembler. If you issue a FILEDEF command using one of the assembler ddnames before you issue the ASSEMBLE command, you can override the default file definitions.

The ddname for the source input file is ASSEMBLE. If you enter:

```
filedef assemble reader
assemble sample
```

then the assembler reads your input file from your card reader, and assigns the file name SAMPLE to the output TEXT and LISTING files. You can use this method to assemble programs directly from DOS sequential files on DOS disks. For example, to assemble a source file named DOSPROG from a DOS disk accessed as a C-disk, you could enter:

```
filedef assemble c dsn dosprog (recfm f lrecl 80
assemble dosprog
```

Again, the name you assign on the ASSEMBLE command may be anything. The assembler uses this name to assign file names to the TEXT and LISTING output files.

LISTING and TEXT are the ddnames assigned to the SYSLST and SYSPCH output of the assembler. You might issue file definitions to override these defaults as follows:

```
filedef listing disk assemble listfile a
filedef text disk assemble textfile a
assemble source
```

When these commands are executed, the output from the assembly of the file SOURCE ASSEMBLE is written to the disk files ASSEMBLE LISTFILE and ASSEMBLE TEXTFILE.

Link-Editing Programs in CMS/DOS

When the assembler or one of the language compilers executes, the object module produced is written to a CMS disk in a file with a file type of TEXT. The file name is always the same as the input source file.

These TEXT files (sometimes referred to as decks, although they are not real card decks) can be used as input to the linkage editor or can be used with an INCLUDE linkage editor control statement.

You can invoke the CMS/DOS linkage editor with the DOSLKED command, for example:

```
doslked test testlib
```

where TEST is the file name of either a DOSLNK or TEXT file (that is, a file with a file type of either DOSLNK or TEXT) or the name of a relocatable module in a system or private relocatable library. TESTLIB indicates the name of the output file which, in CMS/DOS, is a phase library with a file type of DOSLIB.

When you issue the DOSLKED command:

1. CMS first searches for a file with the specified name and a file type of DOSLNK.
2. If none is found, CMS searches the private relocatable library, if you have assigned one. You must issue an ASSGN command for SYSRLB and use the ddname IJSYSRL in a DLBL statement.
3. If the module is still not found, CMS searches all of your accessed disks for a file with the specified name and a file type of TEXT.
4. Last, CMS searches the system relocatable library, if it is available. You must enter the CMS/DOS environment specifying the mode letter of the DOS system residence if you want to access the system libraries.

Linkage Editor Input

You can place the linkage editor control statements ACTION, PHASE, INCLUDE, and ENTRY in a CMS file with a file type of DOSLNK. The DOSLNK file must have fixed-length, 80-byte records. Linkage editor control statements must begin with a blank in column 1 and may not extend beyond column 71.

When you use the INCLUDE statement, you may specify the file name of a CMS TEXT file or the name of a module in a DOS relocatable library:

```
INCLUDE XYZ
```

or you may use the INCLUDE control statement to indicate that the object code follows:

```
INCLUDE
(CMS TEXT file)
```

A typical DOSLNK file, named CONTROL DOSLNK, might contain the following:

```
ACTION REL
PHASE PROGMAIN,S
INCLUDE SUBA
PHASE PROGA,*
INCLUDE SUBB
```

When you issue the command:

```
doslked control
```

the linkage editor searches the following for the object files SUBA and SUBB:

- A DOS private relocatable library, provided you have issued the ASSGN and DLBL commands to identify it:

  ```
  assgn sysrlb d
  dlbl ijsysrl d dsn ? (sysrlb
  ```

- Your CMS disks for files with file names SUBA and SUBB and a file type of TEXT
- The system relocatable library located on the DOS system residence volume (if it is available).

Link-editing TEXT Files

When you want to link-edit individual CMS TEXT files, you can insert linkage editor control statements in the file using the editor and then issue the DOSLKED command:

```
xedit rtnb text
input include rtnc
file
doslked rtnb mydoslib
```

When the above DOSLKED command is executed, the CMS file RTNB TEXT is used as linkage editor input, as long as there is no file named RTNB DOSLNK. The ACTION statement, however, is not recognized in TEXT files.

You can also link-edit relocatable modules directly from a DOS system or private relocatable library, if you have identified the library. If you do this, however, you cannot directly provide control statements for the linkage editor.

To link-edit a relocatable module from a DOS private library and add linkage editor control statements to it, you could use this procedure:

1. Identify the library and use the RSERV command to copy the relocatable module into a CMS TEXT file. In this example, the module RTNC is to be copied from the library OBJ.MODS:

   ```
   assgn sysrlb e
   dlbl ijsysrl e dsn obj mods (sysrlb
   rserv rtnc
   ```

2. Create a DOSLNK file, insert the linkage editor control statements, and copy the TEXT file created in step 1 into it using the GET subcommand:

   ```
   xedit rtnc doslnk
   input action rel
   get rtnc text a
   file
   ```

3. Invoke the linkage editor with the DOSLKED command:

   ```
   doslked rtnc mydoslib
   ```

Alternatively, you could create a DOSLNK file with the following records: DOSLNK file

```
ACTION REL
INCLUDE RTNC
```

and link-edit the module directly from the relocatable library. If you do not need a copy of the module on a CMS disk, you might want to use this method to conserve disk space.

When the linkage editor is reading modules, it may encounter a blank card at the end of a file or a * (comment) card at the beginning of a file. In either case, the linkage editor issues a warning message indicating an invalid card, but it continues to complete the link-edit.

Linkage Editor Output: CMS DOSLIBs

The CMS/DOS linkage editor always places the link-edited executable phase in a CMS library with a file type of DOSLIB. You should specify the file name of the DOSLIB when you enter the DOSLKED command:

```
doslked prog0 templib
```

where PROG0 is the relocatable module you are link-editing and TEMPLIB is the file name of the DOSLIB.

If you do not specify the name of a DOSLIB, the output is placed in a DOSLIB that has the same name as the DOSLNK or TEXT file being link-edited. In the above example, a CMS DOSLIB is created named TEMPLIB DOSLIB, or, if the file TEMPLIB DOSLIB already exists, the phase PROG0 is added to it.

DOSLIBs can contain relocatable core image phases suitable for execution in CMS/DOS. Before you can access phases in them, you must identify them to CMS with the GLOBAL command:

```
global doslib templib permlib
```

When CMS is searching for executable phases, it searches all DOSLIBs specified on the last GLOBAL DOSLIB command. If you have named a number of DOSLIBs or if any particular DOSLIB is very large, the time required for CMS to fetch and execute the phase increases. You should use separate DOSLIBs for executable phases, whenever possible. Then specify only the DOSLIBs you need on the GLOBAL command.

When you link-edit a module into a DOSLIB that already contains a phase with the same name, the directory entry is updated to point to the new phase. However, the space that was occupied by the old phase is not reclaimed. You should periodically issue the command:

```
doslib comp templib
```

to compress the DOSLIB and delete unused space. TEMPLIB is the file name of the DOSLIB.

Linkage Editor Maps

The DOSLKED command also produces a linkage editor map. It writes into a CMS file with a file name specified on the DOSLKED command line and a file type of MAP. The file mode is always A5. If you do not want a linkage editor map, use the NOMAP option on the ACTION statement in a DOSLNK file.

Executing Programs in CMS/DOS

After you have assembled or compiled a source program and link-edited the TEXT files, you can execute the phases in your CMS virtual machine. You may not, however, be able to execute all your DOS programs directly in CMS. There are several execution-time restrictions placed on CMS/DOS programs. You cannot execute a program that uses:

- Multitasking
- More than one partition
- Teleprocessing
- ISAM macros to read or write files
- CMS module files created by GENMOD with the OS option.

Executing DOS Phases

You can load executable phases into your CMS virtual machine using the FETCH command. Phases must be link-edited with ACTION REL before you load them. When you issue the FETCH command, you specify the name of the phase to be loaded:

```
fetch myprog
```

Then you can begin executing the program by issuing the START command:

```
start
```

Or, you can fetch a phase and begin executing it with a single command:

```
fetch prog2 (start
```

When you use the FETCH command without the START option, CMS issues a message telling you at what virtual storage address the phase is loaded:

```
PHASE PROG2 ENTRY POINT AT LOCATION 020000
```

Relocatable phases are always loaded starting at the beginning of the DOS partition unless you specify a different address using the ORIGIN option of the FETCH command:

```
fetch prog3 (origin 22000
start
```

The program PROG3 executes beginning at location X'22000'.

Search Order for Executable Phases

When you execute the FETCH command, CMS searches for the phase name you specify in the following places:

1. In a DOS private core image library on a DOS disk. If you have a private library you want searched for phases, you must identify it using the ASSGN and DLBL commands using the logical unit SYSCLB:

   ```
   assgn sysclb d
   dlbl ijsyscl d dsn ? (sysclb
   ```

   When you enter the DLBL command with the ? operand, you are prompted to enter the DOS file-id.
2. In CMS DOSLIBs on CMS disks. If you want DOSLIBs searched for phases, you must use the GLOBAL command to identify the DOSLIBs to CMS/DOS:

   ```
   global doslib templib mylib
   ```

   You can specify up to 63 DOSLIBs on the GLOBAL command line.
3. On the DOS system residence core image library. If you want the system core image library searched you must have entered the CMS/DOS environment specifying the mode letter of the system residence:

   ```
   set dos on z
   ```

When you want to fetch a core image phase that has copies in both the core image library and a DOSLIB, and you want to fetch the copy from the CMS DOSLIB, you can bypass the core image library by entering the command:

```
assgn sysclb ua
```

When you need to use the core image library, enter:

```
assgn sysclb c
```

where C is the mode letter of the system residence volume. You do not need to reissue the DLBL command to identify the library.

Making I/O Device Assignments

If you are executing a program that performs I/O, you can use the ASSGN command to relate a system or programmer logical unit to a real I/O device:

```
assgn syslst printer
assgn sys052 reader
```

In this example, your program is going to read input data from your virtual card reader. The output print file is directed to your virtual printer. If you want to reassign these units to different devices, you must be sure that the files have been defined as device independent.

If you assign a logical unit to a disk, you should identify the file by using the DLBL command. On the DLBL command, you must always relate the DLBL to the system or programmer logical unit previously specified in an ASSGN command:

```
assgn sys015 b
dlbl myfile b dsn ? (sys015
```

When you enter the DLBL command with the ? operand you are prompted to enter the DOS file-id.

You must issue all of the ASSGN and DLBL commands necessary for your program's I/O before you issue the FETCH command to load the program phase and to begin executing.

Specifying a Virtual Partition Size

For most of the programs that you execute in CMS, you do not need to specify how large a partition you want those programs to execute in. Following the SET DOS ON command, the partition starting address and the size of the partition is described by message DMSSET1101I. If this size (which normally defaults to 100K) is not large enough to contain the program to be executed, you can either let CMS attempt to get additional storage contiguous to the end of the partition or you can use the SET DOSPART command to redefine the size of the partition.

In some instances, you may want to control the partition size:

- For performance considerations
- Because the default may not leave enough free storage to satisfy the GETVIS requests issued by the DOS program or the access method services function being executed.

You can set the partition size with the DOSPART operand of the SET command. For example, after you enter the command:

```
set dospart 60k
```

all programs that you subsequently execute during this session execute in a 60K partition. In this way you can:

- Set a smaller partition size for programs that run better in smaller partitions.
- Set a smaller partition size to leave more free storage. If the reduction of the DOS partition does not free enough storage for the GETVIS requests, a larger virtual machine must be defined. Issuing the command:

  ```
  set dospart off
  ```

  frees the storage allocated to the partition and redefines the partition to the default value. The default size is a maximum of 100K bytes and a minimum of 20K bytes of contiguous storage.

Note: The CMS/DOS partition, unlike partitions under DOS, is used only for the loading and executing of programs invoked by the FETCH or LOAD commands. Areas allocated by GETVIS are assigned addresses outside the partition but within the user's virtual machine.

Setting the UPSI Byte

If your program uses the user program switch indicator (UPSI) byte, you can set it by using the UPSI operand of the CMS SET command. The UPSI byte is initially binary zeros. To set it to ones, enter:

```
set upsi 11111111
```

To reset it to zeros, enter:

```
set upsi off
```

Any value you set remains in effect for the duration of your terminal session unless you reload CMS (with the IPL command).

Debugging Programs in CMS/DOS

You can debug your DOS programs in CMS/DOS using the facilities of CP and CMS. By executing your programs interactively, you can determine the cause of an error or program abend, correct it, and attempt to execute a program again.

Using Exec Procedures in CMS/DOS

During your program development and testing cycle, you may want to create exec procedures to contain sequences of CMS commands that you execute frequently.

For example, if you need a number of MACLIBs, DOSLIBs, and DLBL definitions to execute a particular program, you might have an exec procedure as follows:

```
   /* to set up environment to run program TESTA */

   signal on error
   global maclib testlib dosmac
   assemble testa
   print testa listing
   doslked testa testlib
   global doslib testlib proglib
   access 200 e
   assgn sys010 e
   push dos.test3.stream.beta
   dlbl indd1 e dsn ? '('sys010
   assgn sys011 punch
   cp spool punch to '*'
   assgn sys012 a
```

```
   dlbl outfile a cms test data '('sys012
   signal off error
   fetch testa '('start
   select
     when rc = 100 then do
       .
       .
       .
     end
     when rc = 200 then do
       .
       .
       .
     end
     otherwise
       exit rc
   end

 Error:
   say 'Error occurred on line' sigl':' sourceline(sigl)
   exit rc
```

The ‘signal on error’ control statement in the exec procedure ensures that if an error occurs during any part of the exec, the remainder of the exec does not execute, and the ‘Error’ displays the line number where the error occurred as well as the actual command which gave the error.

Note: For the DLBL command entered with the DSN ? operand, you must stack the response (using ‘push’) before issuing the DLBL command.

When your program is finished executing, the REXX special variable RC indicates the contents of general register 15 at the time the program exited (the ‘Return Code’). You can use this value to perform additional steps in your exec procedure. Additional steps are indicated in the preceding example by ellipses.

Hardware Devices Supported

CMS/DOS routines can read real DOS disks containing VSE data files and VSE private and system libraries. This read support is limited to the following disks supported by VSE:

- IBM 3390 Direct Access Storage
- IBM 9345 Direct Access Storage

Note: If the 3390 Model 9 DASD is used, minidisks used with VSAM are limited to 64K (65,536) tracks. This is a limitation of the VSE/VSAM licensed program. This limit applies whether the minidisk is used by a VSE guest or by CMS/VSAM.

The following devices, which are supported by VSE, are not supported by CMS/DOS:

- Card Readers: 1442, 2560P, 2560S, 2596, 3504, 5425P, and 5425S
- Printers: 2560P, 2560S, 3203 Models 1 and 2, 3525, 5203, 5425P, and 5425S
- Disks: 2311, 3380.

Also, CMS uses the CP spooling facilities and does not support dedicated unit record devices. Each CMS virtual machine supports only one virtual console, one reader, one punch, one printer, four tapes, and 26 disks. Programs that are executed in CMS/DOS are limited to the number of devices supported by CMS.

VSE Supervisor and I/O Macros Supported by CMS/DOS

The programming interfaces defined by the VSE operating system and simulated by CMS are documented below. For definitive information about these interfaces, see the VSE documentation.

CMS/DOS supports the VSE Supervisor macros and the SAM and VSAM I/O macros to the extent necessary to execute the DOS/VS COBOL Compiler, the DOS PL/I Optimizing Compiler, and DOS/VS RPG II Compiler under CMS/DOS. CMS/DOS supports VSE Supervisor macros described in the publication *DOS/VSE Macro Reference*.

CMS does not simulate all VSE operations. Those it does not support are treated as a "no operation" or a "cancel".

The following information deals with the type of support that CMS/DOS provides in the simulation of VSE Supervisor and Sequential Access Method I/O macros. For a discussion of VSAM macros, see “VSE/VSAM Macros Supported” on page 474.

Supervisor Macros

CMS/DOS supports physical IOCS macros and control program function macros for VSE. Table 47 on page 427 lists the physical IOCS macros and describes their support. Table 48 on page 428 lists the control program function macros and their support.

Table 47. Physical IOCS Macros Supported by CMS/DOS

Macro	Support
CCB (command control block)	The CCB is generated. CCW=FORMAT1 is supported only for I/O to the console or to OS or DOS formatted DASD.
IORB (input/output request block)	The IORB is generated. CCW=FORMAT1 is supported only for I/O to the console or to OS or DOS formatted DASD.
EXCP (execute channel program)	The REAL operand is not supported. All other operands are supported.
WAIT	Supported. Issued whenever your program requires an I/O operation (started by an EXCP macro) to be completed before execution of program continues.
SECTVAL (sector value)	Supported for VSAM.
OPEN/OPENR	Supported. Activates a data file.
LBRET (label processing return)	Return to the $$B-transient after an SVC 8 was issued to give control to the problem program.
FEOV (forced end of volume)	Not supported.

Table 47. Physical IOCS Macros Supported by CMS/DOS (continued)

Macro	Support
SEOV (system end of volume)	Not supported.
CLOSE/CLOSER	Supported. Deactivates a data file.

Table 48. SVC Support Routines and Their Operation

Function/Macro	SVC. No. Dec Hex	Support
EXCP	0 0	Used to start an I/O operation on a device in the CMS/DOS environment.
FETCH	1 1	Used to bring a problem program phase into user storage and to start execution of the phase if the phase was found. Operand SYS=YES is not supported.
FETCH	2 2	Used to bring a $$B-transient phase into the CMS transient area (or if the phase is in the CMSDOS segment, not to load it), and start execution of the phase if the phase was found. Operand SYS=YES is not supported.
FORCE DEQUEUE	3 3	Not supported. See note 2.
LOAD	4 4	Used to bring a problem program phase into user storage, and return the caller the entry point address of the phase just loaded. Operand SYS=YES is not supported.
MVCOM	5 5	Provides the user with a means of altering positions 12 through 23 of the partition communications region (BGCOM).
CANCEL	6 6	Cancels a VSE session either by a VSE program request or by a request from any of the CMS routines handling CMS/DOS.
WAIT	7 7	Used to wait on a CCB, IORB, ECB, or TECB. (Note that CMS/DOS does not support ECBs or TECBs). CCBs are always posted by the CMS/DOS routine before returning to the caller. The WAIT support under CMS/DOS will effectively be a branch to the CMS/DOS POST routine.
CONTROL	8 8	Temporarily return control from a $$B-transient to the problem program.
LBRET	9 9	Return to the $$B-transient after an SVC 8 was issued to give control to the problem program.
SET TIMER	10 A	No operation. Successful return code of 0 is given in R15. See note 1.
TRANS. RETURN	11 B	Return from a $$B-transient to the calling problem program.
JOB CONTROL ‘AND’	12 C	Resets flags to 0 in the linkage control byte in BGCOM (communication region). If R1 = 0, bit 5 of JCSW4 (COMREG byte 59) is turned off.
JC FLAGS	13 D	Not supported. See note 2.
EOJ	14 E	Normally terminates execution of a problem program.
SYSIO	15 F	Not supported. See note 2.

Table 48. SVC Support Routines and Their Operation (continued)

Function/Macro	SVC. No. Dec Hex	Support
PC STXIT	16 10	Establish or terminate linkage to a user's program check routine.
PC EXIT	17 11	Used to provide supervisory support for the EXIT macro. SVC 17 provides a return from the user's PC routine to the next sequential instruction in the program that was interrupted because of a program check.
IT STXIT	18 12	No operation. Successful return code of 0 is given in R15. See note 1.
IT EXIT	19 13	Not supported. See note 2.
OC STXIT	20 14	No operation. Successful return code of 0 is given in R15. See note 1.
OC EXIT	21 15	Not supported. See note 2.
SEIZE	22 16	No operation. Successful return code of 0 is given in R15. See note 1.
LOAD HEADER	23 17	Not supported. See note 2.
SETIME	24 18	No operation. Successful return code of 0 is given in R15. See note 1.
HALT I/O	25 19	Not supported. See note 2.
VALID ADDR	26 1A	Validate address limits. The upper address must be specified in general register 2 and the lower address must be specified in general purpose register 1.
TP HALT I/O	27 1B	Not supported. See note 2.
MR EXIT	28 1C	Not supported. See note 2.
WAITM	29 1D	Not supported. See note 2.
QWAIT	30 1E	Not supported. See note 2.
QPOST	31 1F	Not supported. See note 2.
	32 20	Reserved
COMRG	33 21	Used to provide the caller with the address of the partition communications region. CMS/DOS provides the caller with the address of the partition communications region, in the user's register 1.
GETIME	34 24	Provides support for the GETIME macro. SVC 34 updates the date field in the communications region. The GMT operand is not supported.
HOLD	35 23	No operation. Successful return code of 0 is given in R15. See note 1.
FREE	36 24	No operation. Successful return code of 0 is given in R15. See note 1.

Table 48. SVC Support Routines and Their Operation (continued)

Function/Macro	SVC. No. Dec Hex	Support
AB STXIT	37 25	Establish or terminate linkage to a user's abnormal termination routine. Supported for OPTION=DUMP or NODUMP.
ATTACH	38 26	Not supported. See note 2.
DETACH	39 27	Not supported. See note 2.
POST	40 28	Used to post an ECB, IORB, TECB, or CCB. Byte 2, bit 0 of the specified control block is turned 'on' by CMS/DOS.
DEQ	41 29	No operation. Successful return code of 0 is given in R15. See note 1.
ENQ	42 2A	No operation. Successful return code of 0 is given in R15. See note 1.
	43 2B	Reserved
UNIT CHECKS	44 2C	Not supported. See note 2.
EMULATOR INTERF.	45 2D	Not supported. See note 2.
OLTEP	46 2E	Not supported. See note 2.
WAITF	47 2F	Not supported. See note 2.
CRT TRANS	48 30	Not supported. See note 2.
CHANNEL PROG.	49 31	Not supported. See note 2.
LIOCS DIAG.	50 32	Issued by a logical IOCS routine when the LIOCS is called to perform an operation that the LIOCS was not generated to perform. The error message "unsupported function in a LIOCS routine" is issued, and the session is then terminated.
RETURN HEADER	51 33	Not supported. See note 2.
TTIMER	52 34	No operation. Successful return code of 0 is given in R15. See note 1. R0 is also cleared.
VTAM EXIT	53 35	Not supported. See note 2.
FREEREAL	54 36	Not supported. See note 2.
GETREAL	55 37	Not supported. See note 2.
POWER	56 38	Not supported. See note 2.
POWER	57 39	Not supported. See note 2.
SUPVR. INTERF.	58 3A	Not supported. See note 2.
EOJ INTERF.	59 3B	Not supported. See note 2.
GETADR	60 3C	Not supported. See note 2.

Table 48. SVC Support Routines and Their Operation (continued)

Function/Macro	SVC. No. Dec Hex	Support
GETVIS	61 3D	Used to obtain free storage for scratch use or for obtaining an area where a relocatable program may be loaded. The POOL and SVA GETVIS options are ignored. The PAGE option is ignored for requests of less than or equal to 2K bytes of storage. LOC=RES is treated the same as LOC=BELOW.
FREEVIS	62 3E	Used to return the free storage obtained with an earlier GETVIS call.
USE	63 3F	The USE/RELEASE function has been replaced by SVC 110 (LOCK/UNLOCK) for serially controlling system resources. All SVC 63 and 64 requests are mapped into SVC 110 requests respectively. Return codes previously associated with USE/ RELEASE under CMS/DOS are maintained.
RELEASE	64 40	Reference SVC 63.
CDLOAD	65 41	Used to load a relocatable VSAM phase into storage, unless the program has already been loaded.
RUNMODE	66 42	Used by a problem program to find out if the program is running in real or virtual mode. The caller's register 0 is zeroed to indicate that the program is running in virtual mode.
PFIX	67 43	No operation. Successful return code of 0 is given in R15. See note 1.
PFREE	68 44	No operation. Successful return code of 0 is given in R15. See note 1.
REALAD	69 45	Not supported. See note 2.
VIRTAD	70 46	Not supported. See note 2.
SETPFA	71 47	No operation. Successful return code of 0 is given in R15. See note 1.
GETCBUF/ FREECBUF	72 48	Not supported. See note 2.
SETAPP	73 49	Not supported. See note 2.
PAGE FIX	74 4A	Not supported. See note 2.
SECTVAL	75 4B	Used by I/O routines to obtain a sector number for a CKD or ECKD™ device.
SYSREC	76 4C	Not supported. See note 2.
TRANSCCW	77 4D	Not supported. See note 2.
CHAP	78 4E	Not supported. See note 2.
SYNCH	79 4F	Not supported. See note 2.
SETT	80 50	Not supported. See note 2.
TESTT	81 51	Not supported. See note 2.
LINKAGE	82 52	Not supported. See note 2.
ALLOCATE	83 53	Not supported. See note 2.

Table 48. SVC Support Routines and Their Operation (continued)

Function/Macro	SVC. No. Dec Hex	Support
SET LIMIT	84 54	Not supported. See note 2.
RELPAG	85 55	Provides support for the RELPAG macro. At entry register 1 points to a list of 8-byte storage description area. Each entry contains the beginning address and the length-1 of an area to be released. A nonzero byte following an entry indicates the end of the list. An area is released only if it contains at least a full CP page (4K bytes). CMS simulates the release of the pages by setting them to binary zeros. On return, R15 holds return code as follows: **R15 = 0** all areas have been released **R15 = 2** one or more negative area lengths were specified **R15 = 4** one or more pages to be released were outside the user storage area **R15 =16** at least one entry contains a beginning address outside the user storage area.
FCEPGOUT	86 56	No operation. Successful return code of 0 is given in R15. See note 1.
PAGEIN	87 57	No operation. Successful return code of 0 is given in R15. See note 1.
TPIN	88 58	Not supported. See note 2.
TPOUT	89 59	Not supported. See note 2.
PUTACCT	90 5A	Not supported. See note 2.
POWER	91 5B	Not supported. See note 2.
XECBTAB	92 5C	Not supported. See note 2.
XPOST	93 5D	Not supported. See note 2.
XWAIT	94 5E	Not supported. See note 2.
AB EXIT	95 5F	Exit from abnormal task termination routine and continue the task.
TT EXIT	96 60	Not supported. See note 2.
TT STXIT	97 61	Not supported. See note 2.
EXTRACT	98 62	Support for EXTRACT macro of VSE. The caller requests PUB information, CPUID, or storage boundary information. Register 1 on entry points to a parameter list. Output is placed in an area provided by caller.
GETVCE	99 63	Caller requests device information about specific DASD. Information is returned in an output area pointed to from the parameter list. Register 1 contains a pointer to the parameter list on entry.

Table 48. SVC Support Routines and Their Operation (continued)

Function/Macro	SVC. No. Dec Hex	Support
	100 64	Reserved
MODVCE	101 65	No operation. Successful return code of 0 is given in R15. See note 1.
	102 66	Reserved.
SYSFIL	103 67	Not supported. See note 2.
EXTENT	104 68	No operation. Successful return code of 0 is given in R15. See note 1.
SUBSID	105 69	SUBSID.. the ‘INQUIRY’ function is supported for the supervisor subsystem. Information returned is described by the SUPSSID control block. The SUBSID ‘NOTIFY’ and ‘REMOVE’ functions are not supported.
LINKAGE	106 6A	Not supported. See note 2.
TASK INTERF.	107 6B	Provides macro interface support for system information retrieval. The parameters supported are: GETFLD: **field=ppsavar** returns problem program save area address. **=savar** returns current save area address. **=maintask** returns maintask TID in R1. **=aclose** returns in R1: 1 if in process, 0 if not. **=pcexit** returns the pcexit routine address and save area in R0 and R1 respectively. If the exit routine is currently active, bit 0 in R0 is set ON. If no exit is defined, it returns a 0 in both R0 and R1. MODFLD: **field=vsamopen** set bit X'08' in tcbflags byte if R1¬=0 **=aclose** set bit X'10' in tcbflags byte if R1¬=0 The MODFLD requests for fields CNCLALL and OPENSVA are treated as a NOP with a return code of 0. All other SVC 107 macro calls are unsupported. The error message DMSGMF121S is issued and the request is canceled. See note 2.
DATA SECURE	108 6C	Not supported. See note 2.
PAGESTAT	109 6D	Not supported. See note 2.

Table 48. SVC Support Routines and Their Operation (continued)

Function/Macro	SVC. No. Dec Hex	Support
LOCK/UNLOCK	110 6E	Used by VSAM to control access to resources. Access is maintained in either a ‘shared’ or ‘exclusive’ control environment. When DOS is SET ON, counters are maintained and the type of control for each resource in a table (LOCKTAB) built in free storage. All entries not unlocked by the program are cleared at both normal and abnormal end-of-job. All requests for resource control are passed to SVC 110 through the DTL macro (define the lock). SVC 63 and 64 requests are mapped into a dummy DTL and processed by SVC 110.

Note:

1. No operation: In each case, register 15 is cleared to simulate successful operation, and all other registers are returned unchanged, unless otherwise noted.
2. Not supported: For unsupported SVCs, an error message is given, and the SVC is treated as a “cancel”.

Declarative Macros (Sequential Access Method I/O Macros)

CMS/DOS supports the following declarative macros:

- DTFCD - Types X'02' and X'04'
- DTFCN - Types X'03'
- DTFDI - Types X'33'
- DTFMT - Types X'10', X'11', X'12', andX'14'
- DTFPR - Types X'08'
- DTFSD - Types X'20'

The CDMOD, DIMOD, MTMOD, and PRMOD macros generate the logical IOCS routines that correspond with the declarative macros. For files on disk, the logical IOCS routines used during program execution reside in the CMSBAM DCSS and are not generated within the program. The operands that CMS/DOS supports for the DTF are also supported for the xxMOD macro. In addition, CMS/DOS supports three internal macros that the COBOL and PL/I compilers require: DTFCP (types X'31' and X'32'), CPMOD, and DTFSL.

DTFCD Macro — Defines the File for a Card Reader

CMS/DOS does not support the ASOCFLE, FUNC, TYPEFILE=CMBND, and OUBLKSZ operands of the DTFCD macro. CMS/DOS ignores the SSELECT operand and any mode other than MODE=E. Table 49 on page 434 describes the DTFCD macro operands and their support under CMS/DOS. An asterisk (*) in the status column indicates that CMS/DOS support differs from VSE support.

Table 49. CMS/DOS Support of DTFCD Macro

Operand	Status	Description
DEVADDR=SYSxxx		Symbolic unit for reader-punch used for this file.
IOAREA1=xxxxxxxx	*	Name of the first I/O area.
ASOCFLE=xxxxxxxx	*	Not supported.
BLKSIZE=nnn	*	Length of one I/O area, in bytes. If omitted, 80 is assumed. If CTLCHR=YES is specified, BLKSIZE defaults to 81.

Table 49. CMS/DOS Support of DTFCD Macro (continued)

Operand	Status	Description
CONTROL=YES		CNTRL macro used for this file. Omit CTLCHR for this file.
CRDERR=RETRY	*	Retry if punching error is detected. Applies to 2520 only. However, this situation is never encountered under CMS/DOS because hardware errors are not passed to the LIOCS module.
CTLCHR=xxx		(YES or ASA). Data records have control character. YES for S/370 character set; ASA for American National Standards Institute character set. Omit CONTROL for this file.
DEVICE=nnnn	*	(2520, 3505, or 3525).
EOFADDR=xxxxxxxx		Name of your end-of-file routine.
ERROPT=xxxxxx	*	IGNORE, SKIP, or name. Applies to 3505 and 3525 only.
FUNC=xxx	*	Not supported.
IOAREA2=xxxxxxxx	*	If two output areas are used, name of second area.
IOREG=(nn)		Register number if two I/O areas are used and GET or PUT does not specify a work area. Omit WORKA.
MODE=xx	*	Only MODE=E is supported.
MODNAME=xxxxxxxx		Name of the logic module that is used with the DTF table to process the file.
OUBLKSZ=nn	*	Not supported.
RDONLY=YES	*	Causes a read-only module to be generated.
RECFORM=xxxxxx		(FIXUNB, VARUNB, UNDEF). If omitted, FIXUNB is default.
RECSIZE=(nn)	*	Register number if RECFORM=UNDEF.
SEPASMB=YES		DTFCD is to be assembled separately.
SSELECT=n	*	Ignored.
TYPEFLE=	*	Input or output.
WORKA=YES		I/O records are processed in work areas instead of the I/O areas.

DTFCN Macro — Defines the File for a Console

CMS/DOS supports all of the operands of the DTFCN macro. Table 50 on page 435 describes the operands of the DTFCN macro and their support under CMS/DOS. The status column is blank because the CMS/DOS and VSE support of DTFCN are the same.

Table 50. CMS/DOS Support of DTFCN macro

Operand	Status	Description
DEVADDR=SYSxxx		Symbolic unit for the console used for this file.
IOAREA1=xxxxxxxx		Name of I/O area.
BLKSIZE=nnn		Length in bytes of I/O area (for PUTR macro usage, length of output part of I/O area). If RECFORM=UNDEF, maximum is 256. If omitted, 80 is default.

Table 50. CMS/DOS Support of DTFCN macro (continued)

Operand	Status	Description
INPSIZE=nnn		Length in bytes for input part of I/O area for PUTR macro usage.
MODNAME=xxxxxxxx		Logic module name for this DTF. If omitted, IOCS generates a standard name. The logic module is generated as part of the DTF.
RECFORM=xxxxxx		(FIXUNB or UNDEF). If omitted, FIXUNB is default.
RECSIZE=(nn)		Register number if RECFORM=UNDEF. General purpose registers 2 through 12, enclosed in parentheses.
TYPEFLE=xxxxxx		(INPUT, OUTPUT, or CMBND). Input processes both input and output. CMBND must be specified for PUTR macro usage. If omitted, INPUT is default.
WORKA=YES		GET or PUT specifies work area.

DTFDI MACRO — Defines the File for Device Independence for System Logical Units

CMS/DOS supports most operands of the DTFDI macro. Table 51 on page 436 describes the operands of the DTFDI macro and their support under CMS/DOS. An asterisk (*) in the status column indicates that CMS/DOS support differs from VSE support.

Table 51. CMS/DOS Support of DTFDI Macro

Operand	Status	Description
DEVADDR=SYSxxx		(SYSIPT, SYSLST, SYSPCH, or SYSRDR). System logical unit. CMS/DOS issues an error message if the logical unit specified on the DTF does not match the logical unit specified on the corresponding DLBL command.
IOAREA1=xxxxxxxx		Name of the first I/O area.
CISIZE=n	*	This operand specifies the control interval size for a DOS formatted FB-512 device assigned to a nonsystem file logical unit. This operand is ignored for count-key-data devices and CMS formatted disks.
EOFADDR=xxxxxxxx		Name of your end-of-file routine.
FBA=YES		This operand is not required and is ignored if specified.
ERROPT=xxxxxxxx		(IGNORE, SKIP, or name of your error routine). Prevents termination on errors.
IOAREA2=xxxxxxxx		If two I/O areas are used, name of second area.
IOREG2=(nn)		Register number. If omitted and two I/O areas are used, register 2 is default. General purpose registers 2 through 12, enclosed in parentheses.
MODNAME=xxxxxxxx		DIMOD name for this DTF. If omitted, IOCS generates a standard name. This operand is ignored with DASD. The SAM OPEN routines within the CMSBAM DCSS always load an IBM-supplied logic module and link it to the DTF.

Table 51. CMS/DOS Support of DTFDI Macro (continued)

Operand	Status	Description
RDONLY=YES		Generates a read-only module. Requires a module save area for each routine using the module.
RECSIZE=nnn		Number of characters in record. Default values: 121 (SYSLST), 81 (SYSPCH), 80 (other).
SEPASMB=YES		DTFDI to be assembled separately.
TRC=YES	*	Not supported.
WLRERR=xxxxxxxx		Name of your wrong-length record routine.

DTFMT Macro — Defines the File for a Magnetic Tape

CMS/DOS does not support the ASCII, BUFOFF, HDRINFO, LENCHK, and READ=BACK operands of the DTFMT macro. Tape I/O operations are limited to reading in the forward direction.

You may use the FILABL operand in the DTFMT macro to specify that you have a standard tape label file, a nonstandard tape label file, or an unlabeled tape. The type of tape label processing depends on the option selected. See "Label Processing in CMS/DOS" on page 164 for a complete description of tape label processing in CMS/DOS.

Table 52 on page 437 describes the DTFMT macro operands and their support under CMS/DOS. An asterisk (*) in the status column indicates that CMS/DOS support differs from VSE support.

Table 52. CMS/DOS Support of DTFMT Macro

Operand	Status	Description
BLKSIZE=nnnnn		Length of one I/O area in bytes (maximum = 32,767).
DEVADDR=SYSxxx		Symbolic unit for tape drive used for this file.
EOFADDR=xxxxxxxx		Name of your end-of-file routine.
FILABL=xxxx		(NO, STD, or NSTD). If NSTD specified, include LABADDR.
IOAREA1=xxxxxxxx		Name of first I/O area.
ASCII=YES	*	Not supported.
BUFOFF=nn	*	Not supported.
CKPTREC=YES		Checkpoint records are interspersed with input data records. IOCS bypasses checkpoint records.
ERREXT=YES		Additional errors and ERET are desired.
ERROPT=xxxxxxxx		(IGNORE, SKIP, or name of error routine). Prevents job termination on error records.
HDRINFO=YES	*	Not supported.
IOAREA2=xxxxxxxx		If two I/O areas are used, the name of the second area.
IOREG=(nn)		Register number. Use only if GET or PUT does not specify a work area or if two I/O areas are used. Omit WORKA. General purpose registers 2 through 12, enclosed in parentheses.
LABADDR=xxxxxxxx		Name of your label routine if FILABL=NSTD or if FILABL=STD and user-standard labels are processed.

Table 52. CMS/DOS Support of DTFMT Macro (continued)

Operand	Status	Description
LENCHK=YES	*	Not supported.
MODNAME=xxxxxxxx		Name of MTMOD logic module for this DTF. If omitted, IOCS generates standard name.
NOTEPNT=xxxxxx		(YES or POINTS). YES if NOTE, POINTW, POINTR, or POINTS macro is used. POINTS if only POINTS macro is used.
RDONLY=YES		Generate read-only module. Requires a module save area for each routine using the module.
READ=xxxxxxx	*	CMS/DOS only supports READ=FORWARD.
RECFORM=xxxxxx		(FIXUNB, FIXBLK, VARUNB, VARBLK, SPNUNB, SPNBLK, or UNDEF). For work files use FIXUNB or UNDEF. If omitted, FIXUNB is assumed.
RECSIZE=nnnn		If RECFORM=FIXBLK, number of characters in the record. If RECFORM=UNDEF, register number. Not required for other records. General purpose registers 2 through 12, enclosed in parentheses.
REWIND=xxxxxx		(UNLOAD or NORWD). Unload on CLOSE or end-of-volume, or prevent rewinding. If omitted, rewind only.
SEPASMB=YES		DTFMT is to be assembled separately.
TPMARK=NO		Prevent writing a tapemark ahead of data records if FILABL=NSTD or NO.
TYPEFLE=xxxxxx		(INPUT, OUTPUT, or WORK). If omitted, INPUT is default.
VARBLD=(nn)		Register number, if RECFORM=VARBLK and records are built in the output area. General purpose registers 2 through 12 are enclosed in parentheses.
WLRERR=xxxxxxxx		Name of wrong-length record routine.
WORKA=YES		GET or PUT specifies a work area. Omit IOREG.

DTFPR Macro — Defines the File for a Printer

CMS/DOS does not support the ASOCFLE, ERROPT=IGNORE, and FUNC operands of the DTFPR macro. Table 53 on page 438 describes the operands of the DTFPR macro and their support under CMS/DOS. An asterisk (*) in the status column indicates that CMS/DOS support differs from VSE support.

Table 53. CMS/DOS Support of DTFPR Macro

Operand	Status	Description
DEVADDR=SYSxxx		Symbolic unit for the printer used for this file.
IOAREA1=xxxxxxxx		Name for the first output area.
ASOCFLE=xxxxxxxx	*	Not supported.
BLKSIZE=nnn	*	Length of one output area, in bytes. If omitted, 121 is default.
CONTROL=YES		CNTRL macro used for this file. Omit CTLCHR for this file.

Table 53. CMS/DOS Support of DTFPR Macro (continued)

Operand	Status	Description
CTLCHR=xxx		(YES or ASA). Data records have control character. YES for S/370 character set; ASA for American National Standards Institute character set. Omit CONTROL for this file.
DEVICE=nnnn	*	(1403, 1443, 3203, or 3211). If omitted, 1403 is default.
ERROPT=xxxxxxxx	*	RETRY or the name of your error routine for 3211. Not allowed for other devices. IGNORE is not supported.
FUNC=xxxx	*	Not supported.
IOAREA2=xxxxxxxx		If two output areas are used, name of second area.
IOREG=(nn)		Register number; if two output areas used and GET or PUT does not specify a work area. Omit WORKA.
MODNAME=xxxxxxxx		Name of PRMOD logic module for this DTF. If omitted, IOCS generates standard name.
PRINTOV=YES		PRTOV macro used for this file.
RDONLY=YES		Generate a read-only module. Requires a module save area for each routine using the module.
RECFORM=xxxxxx		(FIXUNB, VARUNB, or UNDEF). If omitted, FIXUNB is default.
RECSIZE=(nn)		Register number if RECFORM=UNDEF.
SEPASMB=YES		DTFPR is to be assembled separately.
STLIST=YES		Use 1403 selective tape listing feature.
TRC=YES	*	Not supported.
UCS=xxx		(ON) process data checks. (OFF) ignores data checks. Only for printers with the UCS feature of 3203 or 3211. If omitted, OFF is default.
WORKA=YES		PUT specifies work area. Omit IOREG.

DTFSD Macro — Defines the File for a Sequential DASD

CMS/DOS does not support the FEOVD, HOLD, and LABADDR operands of the DTFSD macro. Table 54 on page 439 describes the operands of the DTFSD macro and their support under CMS/DOS. An asterisk (*) in the status column indicates that CMS/DOS support differs from VSE support.

Table 54. CMS/DOS Support of DTFSD Macro

Operand	Status	Description
BLKSIZE=nnnn		Length of one I/O area, in bytes.
CISIZE=n	*	This operand specifies the control interval size for a DOS formatted FB-512 device assigned to a nonsystem file logical unit. This operand is ignored for count-key-data devices and CMS formatted disks.
EOFADDR=xxxxxxxx		Name of your end-of-file routine.
IOAREA1=xxxxxxxx		Name of first I/O area.
CONTROL=YES		This operand is ignored. CONTROL=YES is always included.

Table 54. CMS/DOS Support of DTFSD Macro (continued)

Operand	Status	Description
DELETFL=NO	*	If DELETFL=NO is specified, the work file is not erased. Otherwise, when the work file is closed, CMS/DOS erases it.
DEVADDR=SYSnnn	*	Symbolic unit. This operand is optional. If DEVADDR is not specified, all I/O requests are directed to the logical unit identified on the corresponding CMS/DOS DLBL command. If a valid logical unit is specified with the DEVADDR operand of the DTF and a different, but also valid, logical unit is specified on the DLBL command, the unit specified on the DLBL command overrides the unit specified in the DTF. However, CMS/DOS issues an error message if a valid logical unit is specified in the DTF and no logical unit is specified on the corresponding DLBL command.
DEVICE=nnnn	*	This operand is ignored. The actual device type is determined by OPEN.
ERREXT=YES		Additional error facilities and ERET are desired. This operand is ignored. ERREXT=YES is always included.
ERROPT=xxxxxxxx		(IGNORE, SKIP, or name of error routine.) Prevents job termination on error records. Do not use SKIP for output files.
FEOVD=YES	*	Not supported.
HOLD=YES	*	Not supported. HOLD=YES is specified for DTFSD update or work files to provide a track hold capability. However, the CMS/DOS open routine sets the track hold bit off and bypasses track hold processing.
IOAREA2=xxxxxxxx		If two I/O areas are used, name of second area.
IOREG=(nn)		Register number. Use only if GET or PUT does not specify work area or if two I/O areas are used. Omit WORKA.
LABADDR=xxxxxxxx	*	Not supported.
MODNAME=xxxxxxxx		This operand is not required. If specified, it is ignored. The SAM OPEN routines within the CMSBAM DCSS always load an IBM-supplied logic module and link it to the DTF.
NOTEPNT=xxxxxxxx		Indicates that NOTE, POINTR, POINTW, and POINTS are used. This operand is ignored. NOTEPNT=YES is always included.
RDONLY=YES		This operand is not required and is ignored if specified. RDONLY=YES is always included.
PWRITE=YES	*	For a DOS-formatted FB-512 disk, this operand specifies that for output operations a physical write occurs for every logical block. This operand is ignored for count-key-data devices and CMS formatted disks. DOS-formatted FB-512 disks are not supported for output.

Table 54. CMS/DOS Support of DTFSD Macro (continued)

Operand	Status	Description
RECFORM=xxxxxx		(FIXUNB, FIXBLK, VARUNB, SPNUNB, SPNBLK, VARBLK, or UNDEF). If omitted, FIXUNB is assumed. For work files, use FIXUNB or UNDEF. Although work files contain fixed-length unblocked records, the CMS file system handles work UNDEF files as variable-length record files. If you specify FIXBLK, VARBLK, or UNDEF when creating a CMS file on a CMS disk, CMS writes the file in variable-length format. The LISTFILE command would show the file as V format. If you specify FIXUNB when creating a CMS file on a CMS disk, CMS writes the file in fixed-length format.
RECSIZE=nnnnn		If RECFORM=FIXBLK, number of characters in record. If RECFORM=SPNUNB, SPNBLK, or UNDEF, register number. Not required for other records.
SEPASMB=YES		DTFSD is to be assembled separately.
TRUNCS=YES		RECFORM=FIXBLK or TRUNC macro used for this file.
TYPEFLE=xxxxxx		(INPUT, OUTPUT, or WORK). If omitted, INPUT is assumed.
UPDATE=YES		Input file or work file is to be updated.
VARBLD=(nn)		Register number if RECFORM=VARBLK and records are built in the output area. Omit if WORKA=YES.
VERIFY=YES		Check disk records after they are written.
WLRERR=xxxxxxxx		Name of your wrong-length record routine.
WORKA=YES		GET or PUT specifies work area. Omit IOREG. Required for RECFORM=SPNUNB or SPNBLK.

Imperative Macros (Sequential Access Method I/O Macros)

CMS/DOS supports the following imperative macros:

- **Initialization macros**: OPEN and OPENR
- **Processing macros**: GET, PUT, PUTR, RELSE, TRUNC, CNTRL, ERET, and PRTOV.

  Note: No code is generated for the CHNG macro.
- **Work file macros for tape and disk**: READ, WRITE, CHECK, NOTE, POINTR, POINTW, and POINTS.
- **Completion macros**: CLOSE and CLOSER.

CMS/DOS supports workfiles containing fixed-length unblocked records and undefined records. Disk work files are supported as single volume, single pack files. Normal extents and split extents are both supported.

EXCP Support in CMS/DOS

CMS/DOS simulates the EXCP (execute channel program) routines to the extent necessary to support the LIOCS routines described in the section, "VSE Supervisor and I/O Macros Supported by CMS/DOS" on page 427.

Because CMS/DOS uses the VSE LIOCS routines, it must simulate all I/O at the EXCP level. The EXCP simulation routines convert all the I/O in the CCW format to CMS physical I/O requests. That is, CMS macros (such as FSREAD/FSWRITE, RDCARD/PUNCHC, PRINTL, and LINERD/LINEWRT) replace the CCW strings. If CMS is performing I/O to DOS disks or to tape, the I/O requests are handled with the DIAGNOSE interface.

When an I/O operation completes, CMS/DOS posts the CCB or IORB with the CMS return code.

CMS/DOS User Considerations and Responsibilities

A critical design assumption of CMS/DOS is that installations that use CMS/DOS for VSE program development also use and have available a VSE system.

You should consider several factors if you plan to use CMS/DOS. The following sections describe some of the user considerations and responsibilities.

VSE System Generation and Updating Considerations

The CMS/DOS support in CMS may use a real VSE system pack. CMS/DOS provides the necessary path and then fetches VSE logical transients and system routines directly as well as the DOS/VS COBOL and DOS PL/I Optimizing compilers from the VSE system or private core image libraries.

It is your responsibility to order a VSE system and then generate it. Also, if you plan to use DOS compilers, you must order the current level of the DOS/VS COBOL compiler and DOS PL/I Optimizing compiler and you must install them on the same VSE system.

When you install the compilers on the VSE system, you must link-edit all the compiler modules as relocatable phases using the following linkage editor control statement:

```
ACTION REL
```

You can place the link-edited phases in either the system or the private core image library.

When you later invoke the compilers from CMS/DOS, the library (system or private) containing the compiler phases must be identified to CMS. You identify all the system libraries to CMS by coding the file mode letter that corresponds to that VSE system disk on the SET DOS ON command when you invoke the CMS/DOS environment. You identify a private library by coding ASSGN and DLBL commands that describe it. The VSE system and private disks must be linked to your virtual machine and accessed before you issue the commands to identify them for CMS.

CMS/DOS has no effect on the update procedures for VSE[19], COBOL, or DOS PL/I. Normal update procedures for applying IBM-distributed coding changes apply.

z/VM Directory Entries

The VSE system and private libraries are accessed in read-only mode under CMS/DOS. If more than one CMS virtual machine is using the CMS/DOS environments you should update the z/VM directory entries so that the VSE system residence volume and the VSE private libraries are shared by all the CMS/DOS users.

The z/VM directory entry for one of the CMS virtual machines should contain the MDISK statements defining the VSE volumes. The z/VM directory entries for the other CMS/DOS users should contain LINK statements.

When the VSE System Must Be Online

Most of what you do in the CMS/DOS environment for VSE program development requires that the VSE system pack or the VSE private libraries be available to CMS/DOS. Usually, you need these VSE volumes whenever:

[19] The CMS/DOS simulation is at the VSE/AF Version 1 level. Version 2 of VSE/AF introduced changes that are incompatible with CMS/DOS. In particular, CMS/DOS commands such as LISTDS, FETCH, or ESERV will not work with a VSE/AF Version 2 SYSRES because of changes made to the internal structure of the VSE/AF library system. Installations which require a VSE/AF SYSRES for the use with CMS/DOS, and for running VSE/AF itself, will have to maintain two SYSRES levels to get full use of both CMS/DOS and VSE/AF.

- You use the DOS/VS COBOL compiler or DOS/PLI Optimizing compiler. The compilers are executed from the system or private core image libraries.
- Your source programs contain COPY, LIBRARY, %INCLUDE, or CBL statements. These statements copy books from your system or from the private source statement library.
- You invoke one of the library programs: DSERV, RSERV, SSERV, PSERV, or ESERV.
- You execute VSE programs that use LIOCS modules. CMS/DOS fetches most of the LIOCS routines for nondisk files directly from VSE system or private libraries.

A VSE system pack is usable when it is:

- Defined for your virtual machine
- Accessed
- Specified, by mode letter, on the SET DOS ON command.

A VSE private library is usable when it is:

- Defined for your virtual machine
- Accessed
- Identified with ASSGN and DLBL commands.

Execution Considerations and Restrictions

The CMS/DOS environment does not support the execution of VSE programs that use:

- Teleprocessing or indexed sequential (ISAM) access methods. CMS/DOS supports only the sequential (SAM) and virtual storage (VSAM) access methods.
- Multitasking. CMS/DOS supports only a single partition, the background partition.

CMS/DOS can be executed in a CMS Batch Facility virtual machine. If any of the VSE programs that are executed in the batch machine read data from the card reader, you must ensure that the end-of-data indication is recognized. Be sure that (1) the program checks for end of data and (2) a /* record follows the last data record.

If there is an error in the way you handle end of data, the VSE program could read the entire batch input stream as its own data. The result is that jobs sent to the batch machine are never executed and the VSE program reads records that are not part of its input file.

Chapter 25. Using Access Method Services and VSAM

This chapter describes how you can use CMS to create and manipulate VSAM catalogs, data spaces, and files on OS and DOS disks using access method services. The CMS support is based on VSE and VSE/VSAM. This means that if you are an OS VSAM user and plan to use CMS to manipulate VSAM files, you are allowed to use those functions of access method services that are available under the access method services portion of VSE/VSAM. The publication*VSE/ESA System Macros User's Guide* describes the control statements you can use.

This chapter also provides information on using the CMS AMSERV command. The CMS AMSERV command lets you execute access method services. Information is provided on using VSAM macros in CMS. The discussion is divided as follows:

- "Using the AMSERV command" contains general information.
- "Manipulating OS and DOS Disks for Use With AMSERV" describes how to use CMS commands with OS and DOS disks.
- "Defining DOS Input and Output Files" is for CMS/DOS users only.
- "Defining OS Input and Output Files" is for OS users only.
- "Using AMSERV Under CMS" includes notes and examples showing how to perform various access method services functions in CMS.
- "Access Method Services Not Supported" lists the functions not supported in CMS.
- "VSE/VSAM Macros" describes the macros and their support in CMS.
- "OS/VSAM Macros" describes the OSVSAM MACLIB supplied with CMS.
- "Hardware Devices Supported" describes the disks supported by VSE that VSAM data sets in CMS can use.

Overview of VSAM under CMS

You can use CMS to:

- Execute the access method services utility programs for VSAM and SAM data sets on OS and DOS disks and minidisks. CMS can both read and write VSAM files using access method services.
- Compile and execute programs that read and write VSAM files from VSE programs.
- Compile and execute programs that read and write VSAM files from OS programs.

VSAM data sets created by CMS are not in the CMS file format. The CMS commands normally used to manipulate CMS files are not applicable to VSAM files. This includes such commands as PRINT, TYPE, XEDIT, COPYFILE, and so on.

VSAM files written by CMS are written using VSE/VSAM. Certain files written under CMS cannot be used directly by OS/VS VSAM. A VSAM data set created by CMS (using VSE/VSAM) has a file format compatible with OS VSAM data sets when the physical record size of the data set is 512, 1K, 2K, or 4K. For more information on the compatibility between VSE/VSAM and OS/VS VSAM files, see the*VSE/ESA General Information* .

Under CMS, VSAM data sets can span up to 25 volumes. CMS does not support VSAM data set sharing. However, CMS supports the sharing (read only) of minidisks or full pack minidisks. The Shared File System does not provide a means of sharing VSAM files.

Because VSAM data sets in CMS are not a part of the CMS file system, CMS file size, record length, and minidisk size restrictions do not apply. The VSAM data sets are manipulated with Access Method Services programs executed under CMS, instead of with the CMS file system commands. Also, all VSAM minidisks and full packs used in CMS must be initialized with the Device Support Facility; the CMS FORMAT command cannot be used.

CMS supports VSAM control blocks with the GENCB, MODCB, TESTCB, and SHOWCB macros.

Executing VSAM Programs under CMS

The commands used to define input and output data sets for Access Method Services (DLBL) and for CMS/DOS users (ASSGN) are also used to identify VSAM input and output files for program execution. Information on executing programs under CMS that manipulate VSAM files is contained in the licensed program documentation for the language processors.

Restrictions on the use of access method services and VSAM under CMS for OS and DOS users are listed in "Access Method Services Not Supported in CMS" on page 472. The *z/VM: CMS Commands and Utilities Reference* contains complete CMS and CMS/DOS command formats, operand descriptions, and responses for each of the commands described in this chapter.

When you are going to execute VSAM programs in CMS or CMS/DOS, you should remember to issue the DLBL commands to identify the master catalog and any other program input or output files you need to define.

Since VSE/VSAM Release 2, VSE/VSAM has reduced its dependency on explicit ASSGN, EXTENT, and DLBL information. In many cases, you no longer need to specify this information. Identification of the master catalog within CMS, however, still requires ASSGN and DLBL commands. For complete information concerning the ASSGN, DLBL, and EXTENT requirements, refer to the*VSE/ESA System Control Statements* .

Note: For ASSGN, EXTENT, and DLBL requirements for multivolume files, refer to "Defining DOS Input and Output Files" on page 453 and "Identifying Existing Multivolume Files" on page 458.

In the discussion that follows, ASSGN, DLBL, and EXTENT information is included even though it may not be required.

Opening an ACB with a MACRF=ADR and subsequently issuing a GET or a PUT with KEYED ACCESS specified in the RPL when SHAREOPTION (4) is specified is not allowed in VSE/VSAM Release 2. Likewise, opening an ACB with KEYED ACCESS and subsequently issuing a GET or a PUT with MACRF=ADR specified in the RPL when SHAREOPTION (4) is specified is not allowed. Please refer to*VSE/ESA System Macros User's Guide* for more information.

VSE/VSAM supports the functions that were previously supported as well as the following enhancements:

- Volume ownership is enhanced so that multiple catalogs may own space in the same DASD volume if only one recoverable catalog owns space on the volume and only if one catalog resides on the volume.
- You can verify the syntax of the AMS commands without actually executing them by using the SYNCHK parameter of the AMS PARM command.
- Using the IGNOREERROR parameter of the AMS DELETE command, you can delete incomplete catalog information that may have resulted from a system failure during DEFINE or DELETE processing. When you specify the IGNOREERROR parameter of the AMS DELETE command, the PRINT option must be used on the CMS AMSERV command to send the listing to the virtual printer.
- By issuing the CMS CATCHECK command, a CMS VSAM user (with or without DOS set ON) may invoke the VSE/VSAM Catalog Check Service Aid to verify a complete catalog structure.
- Using the COMPRESS parameter of the DEFINE function, you can compress or expand data that was previously compressed. This parameter automatically lets VSAM know if the data is to be converted by VSAM when it is read or written.

The AMSERV Command

In CMS, you execute access method services utility programs with the AMSERV command. The basic format is:

```
amserv filename
```

`filename` is the name of a CMS file containing the control statements for access method services.

Note: Throughout the remainder of this section the term AMSERV refers to both the CMS AMSERV command and the OS/VS or VSE/VSAM access method services, except where a distinction is being made between CMS and access method services.

You create an AMSERV file with the CMS editor using a file type of AMSERV and any file name you want. For example:

```
xedit mastcat amserv
```

The editor recognizes the file type of AMSERV and so automatically sets the zone for your input lines at columns 2 and 72. The sample AMSERV file being created in the example above, MASTCAT AMSERV, might contain the following control statements:

```
DEFINE MASTERCATALOG (NAME (MYCAT) -
   VOLUME (123456) CYL(2) -
   FILE (IJSYSCT) )
```

Note: The syntax of the control statements must conform to the rules for access method services, including continuation characters and parentheses. The only difference is that the AMSERV file does not contain a /* for a termination indicator.

Before you can execute the DEFINE control statement in this AMSERV example, you must define the output file, using the ddname IJSYSCT. You can do this using the DLBL command, if required by VSE/VSAM. Since the exact form required in the DLBL command varies according to whether you are an OS or a DOS user, separate discussions of the DLBL command are provided later in this section. All of the following examples assume that any disk data set or file that you are referencing with an AMSERV command was defined by a DLBL command, if required by VSE/VSAM.

When you execute the AMSERV command, the AMSERV control statement file can be on any accessed CMS disk or directory. You do not need to specify the file mode. A CMS/DOS user, unlike a VSE/DOS user, does not need to assign SYSIPT. The task of locating the file and passing it to access method services is performed by CMS.

AMSERV Output Listings

When the AMSERV command is finished processing, you receive the CMS ready message. If there was an error, the return code (from register 15) is displayed following the Ready. For example:

```
Ready(00008);
```

If you are receiving the long form of the ready message, it appears:

```
Ready(00008); T=0.01/0.11 10:50:23
```

If you receive a ready message with an error return code, you should examine the output listing from AMSERV to determine the cause of the error.

AMSERV output listings are written in CMS files with a file type of LISTING. By default, the file name is the same as the input AMSERV file. For example, if you have executed:

```
amserv mastcat
```

and the CMS ready message indicates an error return code, you should examine the file MASTCAT LISTING. Edit the file MASTCAT LISTING and issue the following LOCATE subcommand twice:

```
locate /idc
```

to find the character string IDC will position you in the LISTING file at the first access method services message.

The publication *VSE/VSAM Messages and Codes* lists and explains all of the messages generated by access method services together with the associated return and reason codes.

If you need to make changes to control statements before executing the AMSERV command again, use XEDIT to modify the AMSERV input file.

If you execute the same AMSERV file a number of times, each execution results in a new LISTING file that replaces any previous listing file with the same file name.

Controlling AMSERV Command Listings

When you use AMSERV to print a VSAM file or to list catalog or recovery area contents using the PRINT, LISTCAT, or LISTCRA control statements, the output is written in a listing file on a CMS read/write disk.

If you only want a printed copy of the output listing, issue the AMSERV command with the PRINT option. For example,

```
amserv myfile (print
```

You might want to use this option if you are executing a PRINT or LISTCAT control statement and expect a very large output listing that you know cannot be contained on any of your disks.

If you want to save the output listing on disk and also print a copy, issue the AMSERV command without the PRINT option, and then use the CMS PRINT command to print the LISTING file.

If you issue the AMSERV command with no options, you get a CMS file with a file type of LISTING and a file name equal to the AMSERV input file. This LISTING file is usually written on your A-disk, but if your A-disk is full or not accessed, it is written on any other read/write CMS disk or directory you have accessed.

If there is not enough room on your A-disk or any other disk, the AMSERV command issues an error message saying that it cannot write the LISTING file. If this happens, the LISTING file created may be incomplete and you may not be able to tell whether access method services actually completed successfully. In this case, after you have cleared some space on a read/write disk, you may have to execute an AMSERV PRINT or LISTCAT function to verify the completion of the prior job.

LISTING files take up considerable disk space, so you should erase them as soon as you no longer need them.

Controlling the File Name of the Output Listing

You can also control the file name of the output listing file by specifying a second name on the AMSERV command line. For example,

```
amserv myfile myfile1
```

In this example, the input file is MYFILE AMSERV and the output listing is placed in a file named MYFILE1 LISTING. A subsequent execution of this AMSERV file:

```
amserv myfile myfile2
```

creates a second listing file, MYFILE2 LISTING, so that the listing created from the first execution is not erased.

Calling AMS from an Application Program

Use the NORESET option of the AMSERV command to indicate that the call to the access method services (AMS) is being made dynamically by an application program and that the VSAM environment is not to be reset after the AMS request is complete.

Note that when this option is specified, all VSAM storage will remain allocated until:

- A SET DOS OFF command is issued.
- An EXECOS command is issued.
- An ABEND occurs, in which case the AMSERV environment cannot be reset correctly. If a subsequent AMSERV abends, you must re-IPL CMS.

- The `Ready;` appears.
- Another AMSERV command is issued without the NORESET option.
- A DOS EOJ macro or a DOS CANCEL macro is issued.
- The CATCHECK command is issued.
- Upon completion of an OS/VSAM program call from an exec language program.

This option is most useful when the AMSERV command is issued from an exec or program.

Manipulating OS and DOS Disks for Use with AMSERV

To use CMS VSAM and AMSERV, you can have OS or DOS disks in your virtual machine configuration. They can be assigned in your directory entry, or you can link to them using the CP LINK command. You must have read/write access to them to execute any AMSERV function or VSAM program that requires opening the file for output or update.

Before you can use an OS or DOS disk, you must access it with the CMS ACCESS command:

```
access 200 d
```

The response from the ACCESS command indicates that the disk is in OS or DOS format:

```
D (200) R/W - OS
   — or —
D (200) R/W - DOS
```

You can write on these disks only through AMSERV or through the execution of a program writing VSAM data sets. Once an OS disk is used with AMSERV or VSAM, CMS considers it a DOS disk. Therefore, regardless of whether you are an OS user, when you access or request information about a VSAM disk, CMS indicates that it is a DOS disk. You can still use the disk in an OS or DOS system for VSAM data set processing. Although the format is not changed, the disk is still subject to any incompatibilities that can currently exist between OS and DOS disks.

Data and Master Catalog Sharing

There are two meanings of sharing that must be defined clearly with respect to the CMS support of VSAM. The first is that of the SHAREOPTION parameter found in the DEFINE (and ALTER) command for access method services. (For more information on the DEFINE command, see "The DEFINE and DELETE Functions" on page 467.)

The SHAREOPTION keyword enables the VSAM user to define how a component is shared within or across VSE partitions and VSE systems. Because CMS supports only a single partition environment, cross partition sharing has no meaning in the CMS environment. In addition, because CMS does not provide DASD sharing support, cross system sharing is not supported. Consequently, the SHAREOPTION parameter only has meaning within a CMS virtual machine (functional equivalent of a VSE partition).

The area of sharing most familiar to CMS users is the disk (minidisk) read-sharing provided by CP. For the VSAM user under CMS, it is still possible to share disks in read-only mode in order to read-share VSAM components. However, there is a restriction with respect to the VSAM master catalog. That is, only one virtual machine may have the disk containing the master catalog in write status. This is necessary even if only read functions are being performed during the session. This is due to the master catalog updating read statistics at close time and, when necessary, writing a new control record in the catalog at open time.

In the situation where there is one write status and one or more read-only status, it may be important to ensure data integrity. To make sure that the read-only users get any modified data, use the SHAREOPTION 4 on the DEFINE command. If data integrity is not important, (for example, there will be only a write status at any given time) then use the default SHAREOPTION 1.

Under CMS, it is possible to have the master catalog disk read-only. A programming modification (a bit in the ACB) was made to the DOS/VS VSAM code so that VSAM knows it is running under CMS. If this bit is on, VSAM will not write to the master catalog for either of the two cases described above. This allows one or more CMS virtual machines to share the VSAM master catalog. This assumes either no other virtual machine has the master catalog disk in write status or only one virtual machine (DOS, OS, or CMS) has it.

Multiple CMS users may have the VSAM master catalog disk in read-only status but only one virtual machine may have the same in write status. With respect to data set sharing, there is only read-sharing for the CMS user.

For more information on data and master catalog sharing, see the*VSE/ESA System Macros User's Guide* .

Disk Compatibility

Because the CMS VSAM support writes VSAM data sets to DOS disks, the question of disk compatibility is not one between CMS and DOS nor between CMS or OS but rather between DOS and OS disks. Because CMS actually uses VSE/VSAM for processing VSAM data sets, all disks used by CMS VSAM are DOS disks. For this reason, we need only discuss how DOS and OS disks are compatible.

In the format-4 DSCB, there is a bit in the VTOC indicators (byte 59, bit 0) defined by OS/VS to indicate (when OFF) that a format-5 label is included in the VTOC. This bit is always ON under VSE because DOS does not maintain the format-5 label. This technique allows OS/VS to realize when the format-5 is invalid and that it must recompute free space and rewrite the format-5 label.

Thus, if a disk originally was used under OS/VS, further allocation could occur under VSE but with the format-5 ignored. If the disk was then used under OS/VS and additional allocation performed, OS/VS would recognize the fact that the format-5 was not valid and would rewrite the format-5.

In terms of space allocation, DOS and OS disks are portable between the two systems. However, OS/VS must perform extra processing prior to using the disk if it intends to reallocate using the format-5.

DOS and OS disks containing VSAM data sets are no exception to this. OS and DOS disks containing VSAM data sets that are used under CMS are portable among all three systems. Because CMS uses the actual VSE/VSAM routines, all disks used under CMS to process VSAM data sets become DOS disks.

VSE/VSAM uses physical record sizes ranging from .5K bytes to 8K bytes. All multiples of .5K bytes between those two values are supported. OS/VS VSAM, however, only supports physical record sizes of .5K, 1K, 2K, and 4K. Therefore, some VSAM files written under CMS cannot be used directly by OS/VS VSAM.

Allocating Space

It is necessary to distinguish between two types of allocation under VSAM.

1. The actual space allocation on the disk
2. Allocation within the data set itself

Space for VSAM components must be allocated on the DASD using the DEFINE commands. You can only allocate space for the master catalog, a user catalog, a data space, and a UNIQUE cluster.

In defining the actual DASD space for components, there are parameters for the DEFINE SPACE command that allows the user to include a “secondary allocation” specification. These parameters are CYLINDERS, RECORDS, BLOCKS, and TRACKS. They have this secondary facility only as a syntactic compatibility with the OS/VS access method services commands. That is, VSE (and, therefore, CMS) does not perform secondary space allocation on a DASD.

The facility does exist under VSE (and CMS) to extend data or index components through already allocated data space, catalog extents, or UNIQUE cluster extents. Thus, the CYLINDERS, TRACKS, RECORDS, and BLOCKS parameters of the DEFINE commands for alternate indexes, clusters, and catalogs do not dynamically allocate DASD space but only extend a component through existing space.

Using Minidisks

If you have a minidisk in your virtual machine configuration, you can use it to contain VSAM files. Before you can use it, it must be formatted with the Device Support Facility program. When you request that a disk be added to your user ID for use with VSAM files under CMS, you should indicate that it be formatted for use with OS or DOS. Or you can format it yourself using the Device Support Facility. How to do this is described under "Using Temporary Disks" on page 452.

Note: If you are an OS user, you should be careful about allocating space for VSAM on minidisks. Once you have used CMS AMSERV to allocate VSAM data space on a minidisk, you should not attempt to allocate file space on that minidisk using an OS/VS system. OS does not recognize minidisks, and would attempt to format the entire disk pack and thus erase any data on it. To allocate additional space for VSAM, you should use CMS again.

The LISTDS Command

For OS or DOS disks or minidisks, you can use the LISTDS command to determine the extents of free space available for use by VSAM. You can also determine what space is already in use. You can use this information to supply the extent information when you define VSAM files.

The options used with VSAM disks are:

- EXTENT — to find out what extents are in use
- FREE — to find out what extents are available.

For example, if you have an OS disk accessed as a G-disk, and you enter:

```
listds g (extent
```

The response might look like:

```
EXTENT INFORMATION FOR 'VTOC' ON 'G' DISK:
SEQ TYPE   CYL-HEAD     (RELTRK) TO   CYL-HEAD     (RELTRK)    TRACKS
000 VTOC 00099 00000        1881     00099 00018       1899        19

EXTENT INFORMATION FOR 'PRIVAT.CORE.IMAGE.LIB' ON 'G' DISK: SEQ TYPE
SEQ TYPE   CYL-HEAD     (RELTRK) TO   CYL-HEAD     (RELTRK)    TRACKS
000 DATA 00000 00001           1     00049 00018        949       949

EXTENT INFORMATION FOR 'SYSTEM.WORK.FILE.NO.6' ON 'G' DISK: SEQ TYPE
SEQ TYPE   CYL-HEAD     (RELTRK) TO   CYL-HEAD     (RELTRK)    TRACKS
000 DATA  00050 00000        950     00051 00018        987        38
```

You could also determine the extent for a particular data set:

```
listds ? * (extent
```

CMS responds:

```
DMSLDS220R Enter dataset name:
```

Then, you can enter the file-id:

```
system.recorder.file
```

The response might look like:

```
EXTENT INFORMATION FOR 'SYSTEM RECORDER FILE' ON 'F' DISK:
SEQ TYPE   CYL-HEAD     (RELTRK) TO   CYL-HEAD     (RELTRK)    TRACKS
000 DATA  00102 00000       1938     00102 00018       1956        19
002 DATA  00010 00006        206     00010 00008        208         3
```

LISTDS searches all minidisks accessed until it locates the specified data set. In this example, the data set occupies two separate extents on disk F. If the data set is a multivolume data set, extents on all accessed volumes are located and displayed.

If you want to find the free extents on a particular disk, enter:

```
listds g (free
```

The response might look like:

```
FREESPACE EXTENTS FOR 'G' DISK:
  CYL-HEAD    (RELTRK) TO   CYL-HEAD      (RELTRK)      TRACKS
00052 00000        988    00052 00001         989            2
00054 00002       1028    00080 00000        1520          493
00081 00001       1540    00098 00018        1880          341
```

You can use this information when you allocate space for VSAM files. If you enter:

```
listds * (free
```

CMS lists all the free space available on all of your accessed disks.

Using Temporary Disks

When you need extra space on a temporary basis for use with CMS VSAM and AMSERV, you can use the CP DEFINE command to create a temporary minidisk and then use the Device Support Facilities program to format it. For more information on the Device Support Facility, see the *Device Support Facilities User's Guide and Reference*. Once formatted and accessed, it is available to your virtual machine for the duration of your terminal session or until you detach it using the CP DETACH command. Remember that anything placed on a temporary disk is lost, so that you should copy output that you want to keep onto permanent disks before you log off.

Formatting a Temporary Disk

The example below shows a control statement file and an exec procedure that, together, can be used to format a minidisk using the Device Support Facility. For a complete description of the control statements used, refer to the *Device Support Facilities User's Guide and Reference*.

The input control statements for the Device Support Facility should be placed in a CMS file so that they can be punched to your virtual card reader. For this example, suppose the statements are in a CMS file named TEMP DSF:

```
INIT UNIT(198) DEVTYP(3390) PRG NVFY VOLID(123456) DVTOC(9,7,5) -
  MIMIC (MINI(10))
```

Note: The example above begins in column 2.

Now consider the CMS file named TEMPDISK EXEC:

```
  /* to format a temporary DOS disk */

  signal on error
  cp define t3390 198 10
  cp close reader
  cp purge reader class i
  cp spool punch to '*' class i cont nohold
  punch ipl ickdsf '* ('noh
  punch temp dsf '* ('noh
  cp spool punch nocont close
  cp spool reader class i nohold
  cp ipl 00c clear attn
  exit

  Error:
    exit 100
```

You execute this procedure by entering the file name of the exec:

```
tempdisk
```

When the final line of this exec is executed, the Device Support Facility is in control. You will receive the following three messages:

```
ICK005E DEFINE INPUT DEVICE, REPLY 'DDDD,VDEV OR CONSOLE'
ENTER INPUT/COMMAND:
```

You should enter:

```
2540,00c
```

to indicate that the control statements should be read from your card reader, which is a virtual 2540 device at virtual address 00C.

```
ICK006E DEFINE OUTPUT DEVICE, REPLY 'DDDD,VDEV OR CONSOLE'
ENTER INPUT/COMMAND:
```

You should enter:

```
console
```

to indicate that the utility output should be sent to your console.

```
ICK003D REPLY U TO ALTER VOLUME 198 CONTENTS, ELSE T
ENTER INPUT/COMMAND:
```

You should enter:

```
u
```

to continue the execution.

When the Device Support Facilities program is completed, your virtual machine is in a wait state and you must reload CMS (with the IPL command). You can then access the temporary disk:

```
acc 198 c
```

and CMS responds:

```
C (198) R/W - DOS
```

Defining DOS Input and Output Files

Note: This information is for VSE/VSAM users. OS/VS VSAM users should refer to the section “Defining OS Input and Output Files”.

You may use the DLBL command to define VSAM input and output files for both the AMSERV command and for program execution. The operands required on the DLBL command are:

```
dlbl ddname filemode DSN datasetname (options SYSxxx
```

where ddname corresponds to the *filename* parameter in the AMSERV file and datasetname corresponds to the entry name or file name of the VSAM file.

There are several options you can use when issuing the DLBL command to define VSAM input and output files. These options are:

VSAM
indicates that the file is a VSAM file.

Note: You do not have to use the VSAM option to identify a file as a VSAM file if you are using any of the options:

- EXTENT
- MULT
- CAT
- BUFSP.

These options imply that the file is a VSAM file. In addition, the ddnames (filenames) IJSYSCT and IJSYSUC also indicate that the file being defined is a VSAM file.

EXTENT
defines a catalog or a VSAM data space. You are prompted to enter the volume information. This option provides the function of the EXTENT card in VSE.

MULT
accesses a multivolume VSAM file. You are prompted to enter the extent information.

CAT
identifies a catalog that contains the entry for the VSAM file you are defining.

BUFSP
specifies the size of the buffers VSAM should use during program execution.

Options are entered following the open parenthesis on the DLBL command line, with the SYSxxx:

```
assgn sys003 b
dlbl file1 b dsn workfile (extent cat cat2 sys003
```

Using VSAM Catalogs

While you are developing and testing your VSAM programs in CMS, you may find it convenient to create and use your own master catalog, which may be on a CMS minidisk. VSAM catalogs, like any other cluster, can be shared read-only among several users.

You name the VSAM master catalog for your terminal session using the logical unit SYSCAT in the ASSGN command and the ddname IJSYSCT for the DLBL command. For example, if your VSAM master catalog is located on a DOS disk you have accessed as a C-disk, you would enter:

```
assgn syscat c
dlbl ijsysct c dsn mastcat (syscat
```

Note: When you use the ddname IJSYSCT, you do not need to specify the VSAM option on the DLBL command.

You must define the master catalog at the start of every terminal session. If you are always using the same master catalog, you might include the ASSGN and DLBL commands in an exec procedure or in your PROFILE EXEC. You could also include the commands necessary to access the DOS system residence volume and enter the CMS/DOS environment:

```
ACCESS 350 Z
SET DOS ON Z (VSAM
ACCESS 555 C
ASSGN SYSCAT C
DLBL IJSYSCT C DSN MASTCAT (SYSCAT PERM
```

You should use the PERM option so that you do not have to reset the master catalog assignment after clearing previous DLBL definitions.

You must use the VSAM option on the SET DOS ON command if you want to use any access method services function or access VSAM files.

Defining a Master Catalog

The sample ASSGN and DLBL commands used above are almost identical with those you issue to define a master catalog using AMSERV. The only difference is the EXTENT option that lists the data spaces that this master catalog is to control.

As an example, suppose that you have a 30-cylinder 3390 minidisk assigned to you to use for testing your VSAM programs under CMS. If the minidisk is in your directory at address 333, you should first access it:

```
access 333 d
D (333) R/W - DOS
```

If you formatted the minidisk yourself, you know what its label is. If not, you can find out what the label is by using the CMS command:

```
query search
```

The response might be:

```
USR191  191  A    R/W
DOS333  333  D    R/W - DOS
SYS190  190  S    R/O
SYS19E  19E  Y/S  R/O
```

Use the label DOS333 in the VOLUMES parameter in the MASTCAT AMSERV file:

```
DEFINE MASTERCATALOG -
   (NAME  (MASTCAT ) -
    VOLUME (DOS333) -
    CYL (4) -
    FILE (IJSYSCT)   )
```

To find out what extents on the minidisk you can allocate for VSAM, use the LISTDS command with the FREE option:

```
listds d (free
```

The response from LISTDS might look like this:

```
FREESPACE INFORMATION FOR 'D' DISK:
  CYL-HEAD    (RELTRK) TO   CYL-HEAD      (RELTRK)      TRACKS
00000 00001         1    00000 00009           9           9
00000 00011        11    00029 00018         569         560
```

From this response, you can see that the volume table of contents (VTOC) is located on the first cylinder, so you can allocate cylinders 1 through 29 for VSAM:

```
assgn syscat d
dlbl ijsysct d dsn mastcat (syscat perm extent
DMSDLB331R Enter extent specifications:
19 551
   (null line)
```

After entering the extents, in tracks, giving the relative track number of the first track to be allocated followed by the number of tracks, you must enter a null line to complete the command. A null line is required because, when you enter multiple extents, entries may be placed on more than one line. If you do not enter a null line, the next line you enter causes an error, and you must re-enter all of the extent information.

Note: As in OS, the extents must be on cylinder boundaries, and you cannot allocate cylinder 0.

Now you can issue the AMSERV command:

```
amserv mastcat
```

A ready message with no return code indicates that the master catalog is defined. You do not need to reissue the ASSGN and DLBL commands to use the master catalog for additional AMSERV functions.

Defining User Catalogs

You can use the AMSERV command to define private catalogs and spaces for them. The procedures for determining what space you can allocate are the same as those outlined in the example of defining a master catalog.

To define a user catalog, you may use any programmer logical unit and any ddname:

```
access 199 e
listds e (free
⋮
assgn sys001 e
dlbl cat1 e dsn private.cat1 (sys001 extent perm
⋮
amserv usercat
```

The file USERCAT AMSERV might contain the following:

```
DEFINE USERCATALOG -
     (NAME (PRIVATE.CAT1) -
      CYL (4) -
      VOLUME (DOSVS2) ) -
      CATALOG (MASTCAT)
```

After this AMSERV command has completed successfully you can use the catalog PRIVATE.CAT1. When you issue a DLBL command to identify a cluster or data set cataloged in this catalog, you must identify the catalog using the CAT option on the DLBL command for the file:

```
assgn sys100 c
dlbl file2 c dsn ? (sys100 cat cat1
```

Or, you can define this catalog as a job catalog.

Using Job Catalogs

If you want to set up a user catalog as a job catalog so that it will be searched during all subsequent jobs, you can define the catalog using the special ddname IJSYSUC. For example:

```
assgn sys101 c
dlbl ijsysuc c dsn private.cat1 (sys101 perm
```

If you defined a user catalog (IJSYSUC) for a terminal session and you use the AMSERV command to access a VSAM file, the user catalog takes precedence over the master catalog. This means that for files that already exist, only the job catalog is searched. When you define a cluster, it is cataloged in the job catalog, rather than in the master catalog, unless you use the CAT option to override it.

If you want to use additional catalogs during a terminal session, you first define them just as you would any other VSAM file:

```
assgn sys010 f
dlbl mycat2 f dsn private.cat2 (sys010 vsam
```

Then, when you enter the DLBL command for the VSAM file that is cataloged in PRIVATE.CAT2, use the CAT option to refer to the ddname of the catalog:

```
assgn sys011 f
dlbl input f dsn input.file (sys011 cat mycat2
```

If you want to stop using a job catalog defined with the ddname IJSYSUC, you can clear it using the CLEAR option of the DLBL command:

```
dlbl ijsysuc clear
```

Then, the master catalog becomes the job catalog for files not defined with the CAT option.

Catalog Passwords

When you define passwords for VSAM catalogs in CMS, or when you use CMS to access VSAM catalogs that have passwords associated with them, you must supply the password from your terminal when the AMSERV command executes. The message you receive to prompt you for the password is the same message you receive when you execute access method services:

```
4221A ATTEMPT 1 OF 2. ENTER PASSWORD FOR JOB AMSERV      FILE catalog
```

When you enter the proper password, AMSERV continues execution.

Verifying a Catalog Structure

As a CMS VSAM user (with or without DOS set ON), you can use the CMS CATCHECK command to invoke the VSE/VSAM Catalog Check Service Aid to verify a complete catalog structure. If you do not specify a catalog name with the CATCHECK command, the job catalog specified with the DLBL command is used. CATCHECK produces a print file containing the catalog analysis. For example, issuing:

```
dlbl ijsysuc f dsn private.cat1 (vsam
```

and

```
catcheck
```

results in a print file containing the VSE/VSAM Catalog Check output.

If you had issued only a DLBL for the master catalog, issuing:

```
catcheck private.cat1
```

produces the same result.

Defining and Allocating Space for VSAM files

You can use CMS AMSERV to allocate additional data spaces for VSAM. To use the DEFINE SPACE control statement, you must have defined the catalog that will control the space, and you must have mounted and accessed the volume or volumes where the space is to be allocated.

For example, suppose you have a DOS-formatted 3390 disk attached to your virtual machine at virtual address 255. After accessing the disk and determining the free space on it, you could create a file named SPACE AMSERV:

```
DEFINE SPACE -
   (FILE (FILE1) -
    TRACKS (1900) -
    VOLUME (123456) ) -
    CATALOG (PRIVATE.CAT2 CAT2)
```

Before executing this AMSERV file, define PRIVATE.CAT2 as a user catalog using the ddname CAT2. Then define the ddname for the FILE parameter:

```
access 255 c
assgn sys010 c
dlbl cat2 c dsn private.cat2 (sys010 vsam
assgn sys011 c
dlbl file1 c (extent sys011 cat cat2
amserv space
```

You do not need to enter a data set name to define the space. When CMS prompts you for the extents of the space, you can enter the extent specifications:

```
DMSDLB331R Enter extent specifications:
190 1900
⋮
```

When you define space for VSAM, you should be sure that the VOLUMES parameter and the space allocation parameter (whether CYLINDER, TRACKS, BLOCKS, or RECORDS) in the AMSERV file agree with the information you provide in the DLBL command. All data extents must begin and end on cylinder boundaries. Any additional space you provide in the extent information that is beyond what you specified in the AMSERV file is claimed by VSAM.

Specifying Multiple Extents

When you are specifying extents for a master catalog, data space, or unique file, you can specify up to 16 extents on a volume for a particular space. When prompted by CMS to enter the extents, you must separate different extents by commas or place them on different lines. To specify a range of extents in the above example, you can enter:

```
dlbl file1 c (extent sys011
190  190, 570  190, 1900 1520
   (null line)
  — or —
dlbl file1 c (extent sys011
190  190
570  190
1900 1520
   (null line)
```

Again, the first number entered for each extent represents the relative track for the beginning of the extent and the second number indicates the number of tracks.

Specifying Multivolume Extents

You can define spaces that span up to 25 volumes for VSAM files. All of the volumes must be accessed and assigned when you issue the DLBL command to define or identify the data space.

You should remember, though, that if you are using AMSERV and you do not use the PRINT option, you must have a read/write CMS disk so that AMSERV can write the output LISTING file.

If you are defining a new multivolume data space or unique cluster, you must specify the extents on each volume that the data is to occupy (starting track and number of tracks) followed by the disk mode letter where the disk is accessed and the programmer logical unit to which the disk is assigned. For example:

```
access 135 b
access 136 c
access 137 d
assgn  sys001 b
assgn  sys002 c
assgn  sys003 d
dlbl newfile b (extent sys001
```

If you specify more than one extent on the same line, the extents must be separated by commas. Different extents for the same volume must be entered consecutively.

When you enter multivolume extents, you can use a default mode. For example:

```
dlbl newfile b (extent sys001
DMSDLB331R Enter extent specifications:
100 60, 400 80, 60 40 d sys003,
2000 100 c sys002
   (null line)
```

Any extents you enter without specifying a mode letter and SYSxxx value default to the mode and SYSxxx on the DLBL command line, in this case, the B-disk, SYS001.

If you make any errors issuing the DLBL command or extent information, you must re-enter the entire command sequence.

Identifying Existing Multivolume Files

When you issue a DLBL command to identify an existing multivolume VSAM file, you must use the MULT option of the DLBL command:

```
dlbl old b1 dsn ? (sys002 mult
DMSDLB220R Enter dataset name:
dostest.file
DMSDLB330R Enter volume specifications:
c sys004, d sys003
e sys007
  (null line)
```

When you enter the DLBL command, you should specify the mode letter and logical unit for the first volume on the command line. When you enter the MULT option, you are prompted to enter additional specifications for the remaining extents. In the preceding example, the data set has extents on disks accessed as B-, C-, D-, and E-disks.

Using Tape Input and Output

If you are using AMSERV for a function that requires tape input or output, you must have the tape(s) attached to your virtual machine. The valid addresses for tapes are 181 through 184. When referring to tapes, you can also refer to them using their CMS symbolic names TAP1 through TAP4.

For AMSERV functions that use tape input/output, the TLBL control statement is simulated by building a dummy DLBL containing a user-supplied ddname (filename). CMS does not read tape labels and does not recognize tape data set names.

When you invoke the AMSERV command, you must use the TAPIN or TAPOUT option to specify the tape device being used:

```
amserv export (tapout 181
```

In this example, the output from the AMSERV control statements in a file named EXPORT goes to a tape at virtual address 181. CMS prompts you to enter the ddname:

```
DMSAMS367R Enter tape output DDNAMEs:
```

After you enter the ddname specified on the FILE parameter in the AMSERV file and press the carriage return, the AMSERV command executes.

AMSERV opens all tape files as standard labeled tapes or nonlabeled tapes. If you are using standard labeled tapes, you need to specify a LABELDEF command with AMSERV. The LABELDEF command is the CMS/DOS equivalent of VSE TLBL control statement. The LABELDEF command specifies information in VOL1 and HDR1 labels on the tape. See the description of the LABELDEF command in *z/VM: CMS Commands and Utilities Reference* for more information on this command.

You should use the same name for the *filename* on your LABELDEF command as you do for the ddname you entered in reply to message DMSAMS367R (the ddname specified on the FILE parameter in the AMSERV file). However, the LABELDEF command must be issued before the AMSERV command. The following sequence of commands might be used when you have standard labeled tape output:

```
assgn sys005 tap1
tape rew (181
assgn syscat e
assgn sys006 e
labeldef catout fid catfile volid amserv
dlbl ijsysct e dsn mastcat (syscat vsam
dlbl catin e dsn file (sys006 vsam
amserv repro (tapout 181

DMSAMS367R Enter tape output DDNAMEs:

catout
```

Note: If you do not care what is written in a tape output label or do not want input labels checked, you can specify a LABELDEF with no parameters other than *filename*. When you enter:

```
labeldef intape
```

for an input tape with ddname INTAPE, the standard labels on the tape are skipped without any checking. A similar statement for an output tape writes tape labels with default values (see the description of the LABELDEF command in *z/VM: CMS Commands and Utilities Reference*).

If you use nonlabeled tapes, LABELDEF is not required.

Reading VSAM Tape Files

When you create a tape in CMS using AMSERV, CMS writes a tape mark preceding each output file that it writes. When the same tape is read using AMSERV under CMS, HDR1 and VOL1 labels are checked using the LABELDEF command you provide. If you read this tape in a real VSE system, you should use a TLBL card instead of the LABELDEF command.

Similarly, when you create a tape under a VSE system using access method services, if the tape is created with standard labels, CMS AMSERV has no difficulty reading it.

The only time you should worry about positioning a tape created by AMSERV is when you want to read the tape using a method other than AMSERV, for example, the MOVEFILE command. Then, you must forward space the tape past the label using the CMS TAPE command before you can read it.

Defining OS Input and Output Files

Note: This information is for OS/VS VSAM users only. VSE/VSAM users should refer to “Defining DOS Input and Output Files” for information on defining files for use with VSAM.

The OS/VS VSAM user should bear in mind that CMS uses VSE/VSAM to manipulate VSAM files. The VSAM and AMS statements that can be used are described in the publication *VSE/ESA System Macros User's Guide*.

In addition, there are certain incompatibilities between VSE/VSAM and OS/VS VSAM. For a description of these incompatibilities, refer to the*VSE/ESA General Information* .

If you are going to use access method services to manipulate VSAM or SAM files or you are going to execute VSAM programs under CMS, use the DLBL command to define the input and output files. The basic format of the DLBL command is:

```
DLBL ddname filemode DSN datasetname (options
```

where ddname corresponds to the FILE parameter in the AMSERV file and `datasetname` corresponds to the entry name of the VSAM file. That is, the name specified in the NAME parameter of an access method services control statement.

If you are using a CMS file for AMSERV input or output, use the CMS operand and enter CMS file identifiers as follows:

```
dlbl mine a cms out file1 (vsam
```

The maximum length allowed for ddnames under CMS VSAM is seven characters. This means that if you have assigned eight-character ddnames (or filenames) to files in your programs, only the first seven characters of each ddname are used. So, if a program refers to the ddname OUTPUTDD, you should issue the DLBL command for a ddname of OUTPUTD. Since you can encounter problems with a program that contains ddnames with the same first seven characters, you should recompile those programs using seven-character ddnames.

There are several options you can use when issuing the DLBL command to define VSAM input and output files. These options are:

VSAM
indicates that the file is a VSAM file.

Note: You do not have to use the VSAM option to identify a file as a VSAM file if you are using any of the other options listed here, since they imply that the file is a VSAM file. In addition, the ddnames (filenames) IJSYSCT and IJSYSUC also indicate that the file being defined is a VSAM file.

EXTENT
defines a catalog or a VSAM data space. You are prompted to enter the volume information.

MULT
accesses a multivolume VSAM file. You are prompted to enter the extent information.

CAT
identifies a catalog which contains the entry for the VSAM file you are defining.

BUFSP
specifies the size of the buffers VSAM should use during program execution.

Allocating Extents on OS Disks and Minidisks

When you use access method services to manipulate VSAM files under OS, you do not have to worry about allocating the real cylinders and tracks to contain the files. You can, however, use CMS commands to indicate which cylinders and tracks should contain particular VSAM spaces when you use the DEFINE control statement to define space.

Extents for VSAM data spaces can be defined, in AMSERV files, in terms of cylinders, tracks, or records. Extent information you supply to CMS when executing AMSERV must always be in terms of tracks. When you define data spaces or unique clusters, the extent information (number of cylinders, tracks, or records) in the AMSERV file must match the extents you supply when you issue the DLBL command to define the file. When you supply extent information for the master catalog, any extents you enter in excess of those required for the catalog are claimed by the catalog and used as data space.

CMS does not make secondary space allocation for VSAM data spaces. If you execute an AMSERV file that specifies a secondary space allocation, CMS ignores the parameter.

When you use the DLBL command to define VSAM data space, you can use the EXTENT option indicating to CMS that you are going to enter data extents. For example, if you enter:

```
dlbl space b (extent
```

CMS prompts you to enter the extents:

```
DMSDLB331R Enter extent specifications:
```

When you enter the extents, you specify the relative track number of the first track of the extent, followed by the number of tracks.

You can never write on cylinder 0 track 0, and since VSAM data spaces must be allocated on cylinder boundaries, you should never allocate cylinder 0. Cylinder 0 is often used for the volume table of contents (VTOC) as well. Therefore, it is always best to begin defining space with cylinder 1.

You can determine what disk extents on an OS disk or minidisk are available for allocation by using the LISTDS command with the FREE option, which also indicates the relative track numbers and actual cylinder and head numbers.

Using VSAM Catalogs

While you are developing and testing your VSAM programs in CMS, you may find it convenient to create and use your own master catalog, which may be on a CMS minidisk. VSAM catalogs, like any other cluster, can be shared read-only among several users.

You name the VSAM master catalog for your terminal session using the ddname IJSYSCT for the DLBL command. For example, if your VSAM master catalog is located on an OS disk you have accessed as a C-disk, you would enter:

```
dlbl ijsysct c dsn master catalog (perm
```

You must define the master catalog at the start of every terminal session. If you are always using the same master catalog, you might include the DLBL command you need to define it in your PROFILE EXEC:

```
ACCESS 555 C
DLBL IJSYSCT C DSN MASTCAT (PERM
```

You should use the PERM option so that you do not have to reset the master catalog assignment after clearing previous DLBL definitions. The command:

```
dlbl * clear
```

clears all file definitions except those entered with the PERM option.

Defining a Master Catalog

The sample DLBL command used in the preceding example is almost identical with the one you would issue to define a master catalog using AMSERV. The only difference is that you can enter the EXTENT option so you can list the data spaces that this master catalog is to control. As an example, suppose that you have a 30-cylinder 3390 minidisk assigned to you to use for testing your VSAM programs under CMS. If the minidisk is in your directory at address 333, you should first access it:

```
access 333 d
D (333) R/W - DOS
```

If you formatted the minidisk yourself, you know what label you assigned it. If not, you can find out the label assigned to the disk by issuing the CMS command:

```
query search
```

The response might be:

```
USR191  191  A    R/W
VSAM03  333  D    R/W - DOS
SYS109  190  S    R/O
SYS19E  19E  Y/S  R/O
```

Use the volume label VSAM03 in the MASTCAT AMSERV file:

```
DEFINE MASTERCATALOG -
   (NAME  (MASTCAT ) -
    VOLUME (VSAM03) -
    CYL (4) -
    FILE (IJSYSCT)   )
```

To find out what extents on the minidisk you can allocate for VSAM, use the LISTDS command with the FREE option:

```
listds d (free
```

The response from LISTDS might look like this:

```
FREESPACE INFORMATION FOR 'D' DISK:
CYL-HEAD    (RELTRK) TO   CYL-HEAD      (RELTRK)      TRACKS
00000 00001        1    00000 00009            9           9
00000 00011       11    00029 00018          569         560
```

From this response, you can see that the VTOC is located on the first cylinder, so you can allocate cylinders 1 through 29 for VSAM:

```
dlbl ijsysct d dsn mastcat (perm extent
DMSDLB331R Enter extent specifications:
19 551
    (null line)
```

After entering the extents, in tracks, giving the relative track number of the first track to be allocated followed by the number of tracks, you must enter a null line to complete the command. (A null line is required because, when you enter multiple extents, entries may be placed on more than one line.)

Now you can issue the AMSERV command:

```
amserv mastcat
```

A ready message with no return code indicates that the master catalog is defined. You do not need to reissue the DLBL command to identify the master catalog for additional AMSERV functions.

Defining User Catalogs

You can use the AMSERV command to define private catalogs and spaces for them. The procedures for determining what space you can allocate are the same as those outlined in the example of defining a master catalog. To define a user catalog, you can assign any ddname you want:

```
access 199 e
listds e (free
⋮
dlbl cat1 e dsn private.cat1 (extent
⋮
amserv usercat
```

The file USERCAT AMSERV might contain the following:

```
DEFINE USERCATALOG -
     (NAME (PRIVATE.CAT1) -
      FILE  (CAT1) -
      CYL (4) -
      VOLUME (OSVSAM) ) -
      CATALOG (MASTCAT)
```

After this AMSERV command has completed successfully, you can use the catalog PRIVATE.CAT1. When you define a file cataloged in it, you identify the catalog using the CAT option on the DLBL command:

```
dlbl file2 e dsn ? (cat cat1
```

Or, you can define it as a job catalog.

Using a Job Catalog

During a terminal session, you may be referencing the same private catalog many times. If this is the case, you can identify a job catalog by using the ddname IJSYSUC. Then, that catalog is searched during all subsequent jobs unless you override it using the CAT option when you use the DLBL command to define a file.

If you defined a user catalog (IJSYSUC) for a terminal session and you use the AMSERV command to access a VSAM file, the user catalog takes precedence over the master catalog. This means that for files that already exist, the job catalog is searched. When you define a cluster, it is cataloged in the job catalog, rather than in the master catalog, unless you use the CAT option to override it. CMS never searches more than one VSAM catalog.

You should use the CAT option to name a catalog when the AMSERV file you are executing references, with the CATALOG parameter, a catalog that is not defined either as the master catalog or as a user catalog.

If you want to use additional catalogs during a terminal session, you first define them just as you would any other VSAM file:

```
dlbl mycat2 f dsn private.cat2 (vsam
```

Then, when you enter the DLBL command for the VSAM file that is cataloged in PRIVATE.CAT2 use the CAT option to refer to the ddname of the catalog:

```
dlbl input f dsn input.file (cat mycat2
```

If you want to stop using a job catalog defined with the ddname IJSYSUC, you can clear it using the CLEAR option of the DLBL command:

```
dlbl ijsysuc clear
```

or, you can assign the ddname IJSYSUC to some other catalog. If you clear the ddname for IJSYSUC, then the master catalog becomes the job catalog.

Catalog Passwords

When you define passwords for VSAM catalogs in CMS or when you use CMS to access VSAM catalogs that have passwords associated with them, you must supply the password from your terminal when the AMSERV command executes. The message you receive to prompt you for the password is the same message you receive when you execute access method services:

```
4221A ATTEMPT 1 OF 2. ENTER PASSWORD FOR JOB AMSERV      FILE catalog
```

When you enter the proper password, AMSERV continues execution.

Verifying a Catalog Structure

As a CMS VSAM user (with or without DOS set ON), you can use the CMS CATCHECK command to invoke the VSE/VSAM Catalog Check Service Aid to verify a complete catalog structure. If you do not specify a catalog name with the CATCHECK command, the catalog specified with the DLBL command is used. CATCHECK produces a print file containing the catalog analysis. For example, issuing:

```
dlbl ijsysuc f dsn private.cat1 (vsam
```

and

```
catcheck
```

results in a print file containing the VSE/VSAM Catalog Check output.

If you had issued only a DLBL for the master catalog, issuing:

```
catcheck private.cat1
```

produces the same result.

Defining and Allocating Space for VSAM files

You can use CMS AMSERV to allocate additional data spaces for VSAM. To use the DEFINE SPACE control statement, you must have defined either the master catalog or a user catalog that will control the space, and you must have mounted and accessed the volume or volumes where the space is to be allocated.

For example, suppose you have an OS 3390 disk attached to your virtual machine at virtual address 255. After accessing the disk and determining the free space on it, you could create a file named SPACE AMSERV:

```
DEFINE SPACE -
   (FILE (FILE1) -
    TRACKS (1900) -
    VOLUME (123456) ) -
    CATALOG (PRIVATE.CAT2 CAT2)
```

Before executing this AMSERV file, define PRIVATE.CAT2 using the ddname CAT2. Then define the ddname for the file:

```
access 255 c
dlbl cat2 c dsn private.cat2 (vsam
dlbl file1 c (extent cat cat2
```

You do not need to enter a data set name to define the space. When CMS prompts you for the extents of the space, you can enter the extent specifications:

```
DMSDLB331R Enter extent specifications:
190 1900
⋮
```

When you define space for VSAM, you should be sure that the VOLUMES parameter and the space allocation parameter (whether CYLINDER, TRACKS, BLOCKS, or RECORDS) in the AMSERV file agree with the track information you provide in the DLBL command.

Specifying Multiple Extents

When you are specifying extents for a master catalog, data space, or unique file, you can specify up to 16 extents on a volume for a particular space. When prompted by CMS to enter the extents, you must separate the different extents by commas or place them on different lines. To specify a range of extents in the above example, you could enter:

```
dlbl file1 c (extent
190  190, 570  190, 1900 1520
   (null line)
  — or —
dlbl file1 c (extent
190  190
570  190
1900 1520
   (null line)
```

Again, the first number entered for each extent represents the relative track for the beginning of the extent and the second number indicates the number of tracks.

Specifying Multivolume Extents

You can define spaces that span up to 25 volumes for VSAM files. All of the volumes must be accessed and assigned when you issue the DLBL command to define or identify the data space.

You should remember, though, that if you are using AMSERV and you do not use the PRINT option, you must have a read/write CMS disk so that AMSERV can write the output LISTING file.

If you are defining a new multivolume data space or unique cluster, you must specify the extents on each volume that the data is to occupy (starting track and number of tracks) followed by the disk mode letter at which the disk is assigned:

```
access 135 b
access 136 c
access 137 d
dlbl newfile b (extent
```

If you specify more than one extent on the same line, the extents must be separated by commas. If you enter a comma at the end of a line, it is ignored. Different extents for the same volume must be entered consecutively.

When you enter multivolume extents, you do not have to enter a mode letter for those extents on the disk identified in the DLBL command. For the extents on disk B in the above example, you could enter:

```
dlbl newfile b (extent
DMSDLB331R Enter extent specifications:
100 60, 400 80, 60 40 d
2000 100 c
   (null line)
```

If you make any errors issuing the DLBL command or extent information, you must reissue the entire command sequence.

Identifying Existing Multivolume Files

When you issue a DLBL command to identify an existing multivolume VSAM file, you must use the MULT option of the DLBL command:

```
dlbl old b1 dsn ? (mult
DMSDLB220R Enter dataset name:
vsamtest.file
DMSDLB330R Enter volume specifications:
c, d
```

```
e
   (null line)
```

When you enter the DLBL command you should specify the mode letter for the first disk volume on the command line. When you enter the MULT option you are prompted to enter additional specifications for the remaining extents. In the above example, the data set has extents on disks accessed as B-, C-, D-, and E-disks.

Using Tape Input and Output

If you are using AMSERV for a function that requires tape input or output, you must have the tape(s) attached to your virtual machine. The valid addresses for tapes are 181 through 184. When referring to tapes, you can also refer to them using their CMS symbolic names TAP1 through TAP4.

When you use AMSERV to create or read a tape, you supply the ddname for the tape device interactively, after you issue the AMSERV command. To indicate to AMSERV that you are using tape for input or output, you must use the TAPIN or TAPOUT option to specify the tape device being used:

```
labeldef tapedd fid filename…
amserv export (tapout 181
```

In this example, the output from an EXPORT function is to a tape at virtual address 181. CMS prompts you to enter the ddname:

```
DMSAMS367R Enter tape output DDNAMEs:
```

After you enter the ddname (TAPEDD in this example) for the tape file, AMSERV begins execution.

AMSERV in CMS assumes that tape volumes used for input or output have IBM standard tape labels, for example, VOL1, HDR1, and so forth. The user can override this default by indicating to AMSERV with Access Method Services control statements to use nonlabel tapes. If standard label tapes are used, the LABELDEF command is required. The CMS/DOS routine that performs the tape open needs label information for standard label tapes. See the description of the LABELDEF command in the *z/VM: CMS Commands and Utilities Reference* for further information. The *filename* you specify on the LABELDEF command should be the same one you use to reply to the access method service message that requested you to supply the tape's ddnames. However, the LABELDEF command must be issued before the AMSERV command. If you only want the tape labels skipped, but not checked, enter a LABELDEF with no parameters other than *filename*.

Standard label tapes used for input must always contain standard VOL1, HDR1, and EOF1 labels or they are rejected by CMS AMSERV. Standard label output tapes do not need to contain VOL1 labels because the user is prompted to enter a volume serial number and have the VOL1 label written if he wants. However, blank tapes should not be used for output because the open routine tries to read the tape.

Reading Tapes

When you create a tape file using AMSERV under CMS, CMS writes a label file preceding each output file. When CMS AMSERV reads this file, it checks the HDR1 and VOL1 labels using the LABELDEF command you provide before it reads the data file. If you want to read the tape on a real OS/VS system, however, you must use either LABEL=SL or LABEL=(2,NL) as a parameter on the data definition (DD) card for the tape.

If you are creating a tape under OS/VS access method services to be read by CMS AMSERV, you must be sure to create the tape using standard labels so that CMS can read it properly. CMS cannot read a tape created with LABEL=(,NL) on the DD card.

For CMS to read this tape for any other purpose (for example, to use the MOVEFILE command to copy it), you must remember to forward space the file past the tape mark before beginning to read it.

Using AMSERV under CMS

This section provides examples of AMSERV functions executed under CMS. The examples are applicable to both the CMS (OS) and CMS/DOS environments. You should be familiar with the material presented in either "Defining DOS Input and Output Files" on page 453 or "Defining OS Input and Output Files" on page 460, depending on whether you are a DOS or an OS user, respectively. For the examples shown below, command lines and options that are required only for CMS/DOS users are shaded. OS users should ignore these shaded entries.

A CMS variable format file cannot be used directly as input to AMSERV functions as a variable (V) or variable blocked (VB) file because the standard variable CMS record does not contain the BL and RL headers needed by the variable record modules. If these headers are not included in the record, errors will result.

All files placed on the CMS disk by AMSERV show a RECFM of V, even if the true format is fixed (F), fixed blocked (FB), undefined (U), variable (V), or variable blocked (VB). You must know the true format of the file you are trying to use with the AMSERV command and access it properly or errors will result.

A CMS standard variable-format file can be accessed as RECFM=U to use the file as follows:

```
AMSERV  AMREPUV
```

The file AMREPUV AMSERV contains the following 2 cards:

```
REPRO  INFILE (INPUT ENV(RECFM(U),BLKSZ(800),PDEV(3390)))
       OUTFILE (OUTPUT ENV(RECFM(V),BLKSZ(800),RECSZ(84),PDEV(3390)))
```

The input file can be any CMS file with LRECL 800 or less. The output file will be a true variable file that can be used with AMSERV.

The DEFINE and DELETE Functions

When you use the DEFINE and DELETE control statements of AMSERV, you do not need to specify the DSN parameter on the DLBL command:

```
assgn syscat c
dlbl ijsysct c (perm extent syscat
```

If the above commands are executed prior to an AMSERV command to define a master catalog, the DEFINE will be successful if you have assigned a data set name using the NAME parameter in the AMSERV file. The same is true when you define clusters or when you use the DELETE function to delete a cluster, space, or catalog.

When you do not specify a data set name, AMSERV obtains the name from the AMSERV file. In the case of defining or deleting space, no data set name is needed. The FILE parameter corresponding to the ddname is all that is necessary, and AMSERV assigns a default data set name to the space.

When you define space on a minidisk using AMSERV, CMS does not check the extents you specify to see whether they are greater than the number of cylinders available. As long as the starting cylinder is a valid cylinder number and the extents you specify are on cylinder boundaries, the DEFINE function completes successfully. However, you receive an error message when you use an AMSERV function that tries to use this space.

Defining a Suballocated Cluster

To define a cluster for VSAM space that has already been allocated, you need:

1. An AMSERV file containing the control statements necessary for defining the cluster, and
2. The master catalog (and, perhaps, user catalog) volume, which will point to the cluster.

The volume where the cluster is to reside does not have to be online when you define a suballocated cluster.

For example, the file CLUSTER AMSERV contains the following:

```
DEFINE CLUSTER  (  NAME (BOOK.LIST) -
      VOLUMES (123456) -
      TRACKS (40) -
      KEYS (14,0) RECORDSIZE (120,132)  ) -
    DATA (NAME (BOOK.LIST.DATA) ) -
    INDEX (NAME (BOOK.LIST.INDEX) )
```

To execute this file, you would need to enter the following command sequence (assuming that the master catalog, on volume 123456, is in your virtual machine at address 310):

```
access 310 b
assgn syscat b
dlbl ijsysct b (perm syscat
amserv cluster
```

Defining a Unique Cluster

For a unique cluster (one defined with the UNIQUE attribute), you must define the space for the cluster at the same time you define its name and attributes. Therefore, the volume or volumes where the cluster is to reside must be mounted and accessed when you execute the AMSERV command. You can supply extent information for the cluster's data and index portions separately.

Suppose UNIQUE AMSERV contains the following (the ellipses indicate that the AMSERV file is not complete):

```
DEFINE CLUSTER -
      (NAME (PAYROLL) ) -
      DATA ( FILE (UDATA) -
             UNIQUE -
             VOLUMES (567890) -
             CYLINDERS (40) -
             … ) -
      INDEX ( FILE (UINDEX) ) -
             UNIQUE -
             VOLUMES (567890) -
             CYLINDERS (10) -
             … )
```

To execute UNIQUE AMSERV, issue the following command sequence:

```
access 350 c
assgn sys004 c
dlbl udata c (extent sys004
DMSDLB331R Enter extent specifications:
800 800 c sys004
dlbl uindex c (extent sys004
600 200 c sys004
amserv unique
```

Deleting Clusters, Spaces, and Catalogs

When you use AMSERV to delete a VSAM cluster, the volume containing the cluster does not have to be accessed unless the volume also contains the catalog where the cluster is defined. In the case of data spaces and user catalogs or the master catalog, the volume(s) must be mounted and accessed to delete the space.

When you delete a cluster or a catalog, you do not need to use the DLBL command, except to define the master catalog; AMSERV can obtain the necessary file information from the AMSERV file.

When you are using temporary disks with AMSERV, you should be particularly careful that you have not cataloged a temporary data space or cluster in a permanent catalog. You will not be able to delete the space or cluster from the catalog.

Using Data Compression Services

Both CMS and GCS support the VSE/VSAM for VM Version 6 Release 1 Data Compression Services, which allows automatic compression and expansion of data records on clusters DEFINEd as COMPRESS. ESDS, KSDS, and VRDS clusters can then be defined as COMPRESS and related to the compression control data set in the catalog where the cluster is defined. RRDS and SAM-ESDS cluster types cannot be defined as COMPRESS. Note that clusters defined as COMPRESS type cannot be opened in control interval mode. Compressed data is under the control of the VSE/VSAM program.

Creating a Compressed CLUSTER

Two things must be done to create a new data set in compressed format:

1. A 'VSAM.COMPRESS.CONTROL' KSDS compression control data set must be defined in each catalog where compressed data will reside.
2. The new data set CLUSTER must be defined as COMPRESS format.

When you use AMSERV to create a VSAM cluster, the COMPRESS parameter of the DEFINE function will allow record data to be compressed when it is written and will expand data when it is read. This parameter automatically lets VSAM know if the data is to be converted by VSAM when it is read or written; no application program changes are necessary.

Application Migration Considerations

An existing application can take advantage of these VSAM Data Compression Services without the need for program changes. The compression controls are in the VSAM product and are not tied to the application code. Two things must be done to migrate existing data sets to compressed format:

1. A 'VSAM.COMPRESS.CONTROL' KSDS compression control data set must be defined in each catalog where compressed data will reside.
2. The existing data set CLUSTER must be redefined as COMPRESS format.

Use the following examples to assist you in migrating to a compressed format.

Compression control data set example

```
DEFINE CLUSTER (NAME(VSAM.COMPRESS.CONTROL)
                RECORDS(200 100)
                SHAREOPTIONS(4 4)
                RECSZ(80 500)
                VOL(123456)
                NOREUSE
                NOIMBED
                INDEXED
                FREESPACE(15 5)
                KEYS(44 0)
                TO(99366))
       DATA (NAME(VSAM.COMPRESS.CONTROL.@D@)
                CNVSZ(512))
       INDEX (NAME(VSAM.COMPRESS.CONTROL.@I@))
       CATALOG(MASTCAT)
```

KSDS with the COMPRESS parameter example

```
DEFINE CLUSTER (NAME(KSDS1)
                TRK(5 5)
                FILE(KSDS01)
                VOL(123456)
                KEYS(2 0)
                CNVSZ(512)
                COMPRESS
                RECSZ(80 80))
       DATA (NAME(KSDS1.DATA))
       INDEX (NAME(KSDS1.INDEX))
       CATALOG(MASTCAT)
```

Existing data sets can be unloaded temporarily so that the cluster can be redefined as compressed. The cluster can then be reloaded to create the compressed database which is immediately usable by application programs.

For more information on VSE/VSAM Data Compression Services, see *VSE/VSAM Version 6 Release 1 Commands* and *VSE/VSAM Version 6 Release 1 User's Guide and Application Programming*.

The REPRO, IMPORT, and EXPORT Functions

You can manipulate VSAM files in CMS with the REPRO, IMPORT, and EXPORT functions of AMSERV. You can create VSAM files from sequential tape or disk files (on OS, DOS, or CMS disks) using the REPRO function. Using REPRO, you can also copy VSAM files into CMS disk files or onto tapes. For the IMPORT/EXPORT process, you have the option (for smaller files) of exporting VSAM files to CMS disks or to tapes.

You cannot, however, use the EXPORT function to write files onto OS or DOS disks. Nor can you use the REPRO function to copy ISAM (indexed sequential) files into VSAM data sets, because CMS cannot read ISAM files.

When creating a VSAM file from a non-VSAM disk file, the device track size must be the maximum BLOCKSIZE in the INFILE statement. AMSERV expects a DOS or OS file as input and will not open a disk file when the BLOCKSIZE specified is greater than the track capacity of the disk device being used.

You cannot use the ERASE or PURGE options of the EXPORT command if you are exporting a VSAM file from a read-only disk. The export operation succeeds, but the listing indicates an error code 184, meaning that the erase function could not be performed.

You should not use an EXPORT DISCONNECT function from a CMS minidisk and try to perform an IMPORT CONNECT function for that data set onto an OS system. OS incorrectly rebuilds the data set control block (DSCB) that indicates how much space is available.

Copying a CMS Sequential File into a VSAM File

The AMSERV file below gives an example of using the REPRO function to copy a CMS sequential file into a VSAM file. The CMS input file must be sorted in alphanumeric sequence before it can be copied into the VSAM file, which is a keyed sequential data set (KSDS). The VSAM cluster, NAME.LIST, is defined in an AMSERV file named PAYROLL:

```
DEFINE CLUSTER ( NAME (NAME.LIST  ) -
       VOLUMES (CMSDEV) -
       TRACKS (20) -
       KEYS (14,0) -
       RECORDSIZE (120,132) ) -
    DATA (NAME (NAME.LIST.DATA) ) -
    INDEX (NAME (NAME.LIST.INDEX ) )
```

To sort the CMS file, create the cluster, and copy the CMS file into it, use the following commands:

```
sort name list a name sort a
DMSSRT604R Enter sort fields:
1 14
access 135 c
assgn syscat c
dlbl ijsysct c (perm syscat
amserv payroll
assgn sys006 a
dlbl sort a cms name sort (sys006
assgn sys007 c
dlbl name c dsn name list (sys007 vsam
amserv repro
```

The file REPRO AMSERV contains:

```
REPRO   INFILE ( SORT  -
          ENV (RECORDFORMAT (F) -
               BLOCKSIZE (80) -
               PDEV (3390) ) ) -
         OUTFILE (NAME)
```

EXPORTing a VSAM File to a Tape

When you use the REPRO, IMPORT, or EXPORT functions with tape files, you must remember to use the TAPIN and TAPOUT options of the AMSERV command. These options perform two functions:

- They allow you to specify the device address of the tape.
- They notify AMSERV to prompt you to enter a ddname.

In the example below, a VSAM file is being exported to a tape. The file, TEXPORT AMSERV, contains:

```
EXPORT  NAME.LIST -
        INFILE (NAME) -
        OUTFILE (TAPE ENV (PDEV (2400) ) )
```

To execute this AMSERV, you enter the commands as follows:

```
assgn sys006 c
dlbl name c (sys006 vsam
amserv texport (tapout 181
DMSAMS367R Enter tape output DDNAMEs:
tape
```

Writing Execs for AMSERV and VSAM

You may find it convenient to use exec procedures for most of your AMSERV functions, as well as setting up input and output files for program execution, and executing your VSAM programs. If, for example, a particular AMSERV function requires several disks and a number of DLBL statements, you can place all of the required commands in an exec file.

Suppose you have the following file called SETUP EXEC:

```
/*   */
ACCESS 135 B
ACCESS 136 C
ACCESS 137 D
ACCESS 300 G
ASSGN SYSCAT G
DLBL IJSYSCT G '('PERM SYSCAT
ASSGN SYS001 B
DLBL FILE1 B DSN FIRST FILE '('VSAM SYS001
ASSGN SYS002 C
DLBL FILE2 C DSN SECOND FILE '('VSAM SYS002
ASSGN SYS003 D
DLBL FILE3 D DSN THIRD FILE '('VSAM SYS003
AMSERV MULTFILE
```

To invoke this sequence of commands, enter the name of the exec:

```
setup
```

If you place the following statement at the beginning of the exec file:

```
signal on error
```

and then place the following statements at the end of the exec:

```
Error:
Say 'Unexpected return code' rc 'from command'
Say sourceline(sigl) 'at line' sigl'.'
Exit rc
```

you can be sure that the AMSERV command does not execute unless all of the prior commands completed successfully.

For those AMSERV functions that issue response messages, you can use the REXX PUSH or QUEUE instructions. For example:

```
/* An exec to invoke AMSERV */
signal on error
access 305 d
assgn sys007 d
dlbl output d '('vsam sys007
labeldef tape fid file1
signal off error
push tape
amserv timport '('tapin 181
if rc ¬= 0
  then type timport listing
tape rew
exit 0

Error: Say 'Unexpected return code' rc 'from command' Say
sourceline(sigl) 'at line' sigl'.' Exit rc
```

When the AMSERV command in the exec is executed, the request for the tape ddname is satisfied immediately by the response stacked with the PUSH statement.

If you are executing a command that accepts multiple response lines, you have to stack a null line as follows:

```
 queue c sys002',' d sys003
 queue
 dlbl multfile b '('mult sys001
```

Successive iterations of programs that call the OS environment with VSAM result in an abend and error message DMSIPT141T. To avoid this problem, the user should include an EXECOS at the beginning of the exec to clear and reset the OS and VSAM environment without having to return to the interactive environment.

VSE/VSAM Functions Not Supported in CMS

Refer to the publication *Using VSE/VSAM Commands and Macros* for a description of Access Method Services functions available under VSE, and, therefore, under CMS. This knowledge of Access Method Services is assumed throughout this publication.

All of VSE/VSAM is supported by CMS, except for the following:

- Non-VSAM data sets with data formats that are not supported by CMS/DOS (for example, BDAM and ISAM files are not supported).
- The SHAREOPTIONS operand is not supported for cross system or cross partition sharing in CMS/DOS (that is, DASD sharing is not supported).
- The EXCEPTIONEXIT operand of DEFINE CLUSTER or DEFINE ALTERNATE INDEX is not supported under CMS (both CMS/DOS and OS users).
- Space Management for SAM Feature
- Backup/Restore Feature.

If an AMSERV input file to VSE/VSAM Access Method Services contains the control statement *DELETE* with *IGNORERROR*, the PRINT option on the AMSERV command must be used to send the output to the virtual printer.

Execution of VSAM macros from a nucleus extension is not supported by CMS.

Access Method Services Not Supported in CMS

In CMS, an OS user is a user that has ***not*** issued the command:

```
SET DOS ON (VSAM)
```

OS users can use all of the Access Method Services functions that are supported by VSE/VSAM, with the following exceptions:

- Non-VSAM data sets with data formats that are not supported by CMS/DOS (for example, BDAM and ISAM files are not supported).
- The SHAREOPTIONS operand is not supported for cross system or cross partition sharing in CMS/DOS (that is, DASD sharing is not supported).
- The EXCEPTIONEXIT operand of DEFINE CLUSTER or DEFINE ALTERNATE INDEX is not supported under CMS (both CMS/DOS and OS users).
- Do not use the AUTHORIZATION (entrypoint) operand in the DEFINE and ALTER commands unless your own authorization routine exists on the DOS core image library, the private core image library, or in a CMS DOSLIB file. In addition, results are unpredictable if your authorization routine issues an OS SVC instruction.
- The OS Access Method Services GRAPHICS TABLE options and the TEST option of the PARM command are not supported.
- The file name in the FILE (filename) operands is limited to seven characters. If an eighth character is specified, it is ignored.
- The OS access method services CNVTCAT and CHKLIST commands are not supported in VSE/VSAM access method services. In addition, all OS access method services commands that support the 3850 Mass Storage System are not supported in DOS/VS access method services.
- Table 55 on page 473 is a list of OS operands, by control statement, that are not supported by the CMS interface to VSE/VSAM Access Method Services.

If any of the unsupported operands or commands in Table 55 on page 473 are specified, the AMSERV command terminates and displays an appropriate error message.

Table 55. OS Access Method Service Operands NOT Supported in CMS

OS ACCESS METHOD SERVICES CONTROL STATEMENT	**OPERANDS NOT SUPPORTED IN CMS**
ALTER	EMPTY/NOEMPTY SCRATCH/NOSCRATCH DESTAGEWAIT/NODESTAGEWAIT STAGE/BIND/CYLINDERFAULT
DEFINE	ALIAS EMPTY/NOEMPTY GENERATIONDATAGROUP PAGESPACE SCRATCH/ NOSCRATCH DESTAGEWAIT/NODESTAGEWAIT STAGE/BIND/CYLINDERFAULT TO/FOR/OWNER[20]
DELETE	ALIAS GENERATIONDATAGROUP PAGESPACE
EXPORT	OUTDATASET
IMPORT	INDATASET OUTDATASET IMPORTA
LISTCAT	ALIAS GENERATIONDATAGROUP LEVEL OUTFILE[21] PAGESPACE
PRINT	INDATASET OUTFILE[21]
REPRO	INDATASET OUTDATASET

When you use the PRINT, EXPORT, IMPORT, IMPORTRA, EXPORTRA, and REPRO control statements for sequential access method (SAM) data sets, you must specify the ENVIRONMENT operand with the required DOS options (that is, PRIME DATA DEVICE, BLOCKSIZE, RECORDSIZE, or RECORDFORMAT). You must have previously issued a DLBL for the SAM file.

[20] The TO/FOR/OWNER operands are supported for the access method services interface, but are not supported for the DEFINE NONVSAM control statement.

[21] The OUTFILE operand is supported by the access method services interface, but is not supported for the LISTCAT and PRINT control statements.

AMSERV can write SAM data sets only to a CMS disk or directory, but can read them from DOS, OS, or CMS disks or directories.

ISAM Interface Program (IIP)

CMS does not support the VSAM ISAM Interface Program (IIP). Thus, any program that creates and accesses ISAM (indexed sequential access method) data sets cannot be used to access VSAM key sequential data sets. There is one exception to this restriction. You can execute VSAM I/O requests if:

1. Your OS PL/I programs have files declared as ENV(INDEXED)
2. The library routines detect that the data set being accessed is a VSAM data set.

VSE/VSAM Macros Supported

The programming interfaces defined by the VSE operating system and simulated by CMS are documented below. For definitive information about these interfaces, see the VSE/VSAM documentation.

The VSE/VSAM macros and their options are supported for use in assembler language programs under CMS/DOS. The VSE/VSAM macros are:

ACB	EXLST	SHOWCAT
BLDVRP	GENCB	SHOWCB
DLVRP	MODCB	TCLOSE
ENDREQ	POINT	TESTCB
ERASE	RPL	WRTBFR

The VSE/VSAM macros are distributed with the VSE/VSAM product.

All options are supported with the exception of “AM=VSAM”. This option is not supported on any of the macros.

The EXLST EXCPAD exit may be specified, but it is never taken in the CMS environment. The reason is that VSE/VSAM takes this exit when it is waiting for I/O to complete, but in the CMS environment, I/O is always complete when control is returned to VSE/VSAM.

In addition to the above list of macros, the following list of VSE macros normally used with the VSAM macros are also supported. The following macros are distributed with CMS for use with VSAM only.

Table 56. VSE Macros Normally Used with VSAM Macros

VSE macro Supported	**Extent of Support**
CDLOAD	Only supported to the extent required for VSAM execution.
CLOSE	Supported for both VSAM and SAM.
CLOSER	Supported for both VSAM and SAM.
GET	Supported for both VSAM and SAM.
OPEN	Supported for both VSAM and SAM.
OPENR	Supported for both VSAM and SAM.
PUT	Supported for both VSAM and SAM.

VSE Supervisor Macros and Logical Transients Support

VSE supervisor macros required by VSE/VSAM are supported by CMS. See Table 46 on page 418 for a complete list of supervisor macros supported.

CMS distributes the VSE transients that are needed in the VSAM support. Thus, OS users do not need to have the VSE system pack online when they are compiling and executing VSAM programs.

CMS uses all of the VSE B-transients except those that build and release extent blocks. The extent block is not supported in CMS and, thus, neither are the B-transients that control extent blocks.

The CMSDOS shared segment contains the B-transients that are simulated for VSE support in CMS. The B-transients pertaining only to VSAM are included in the VSAM saved segment. Other VSE routines required by VSE/VSAM are contained in the CMSBAM shared segment. This includes the common VTOC handler routines, SAM data management, and the VSAM look-aside function.

OS/VSAM Macros Supported in CMS

A subset of the OS/VSAM macros is supported for use in CMS. The macros are at an MVS DFP 2.3.0 level and are contained in the OSVSAM MACLIB shipped with z/VM.

The macros provide compatibility with the MVS/XA 2.3.0 24-bit or 31-bit addressing Application Programming Interface (API), but have no effect on execution. The macros are:

ACB	ERASE	MODCB	RPL
CHECK	EXLST	OPEN	SHOWCB
CLOSE	GENCB	POINT	TESTCB
ENDREQ	GET	PUT	

VSAM Macro Options Not Supported in CMS

Some options of the OS/VSAM macros do not work in CMS because OS/VSAM macro requests are executed using VSE/VSAM code. Table 57 on page 475 lists the OS/VSAM macros with **unsupported** options.

Table 57. Unsupported Options of OS/VSAM Macros

OS/VSAM Macro	**Unsupported Options**
ACB	MACRF=CFX, NFX, ICI, NCI, LSR, GSR, DFR, NIS, SIS
	SHRPOOL
EXLST	UPAD
GENCB BLK=ACB	MACRF=CFX, NFX, ICI, NCI, LSR, GSR, DFR, NIS, SIS
	SHRPOOL
GENCB BLK=EXLST	UPAD
GENCB BLK=RPL	TRANSID
MODCB ACB	MACRF=CFX, NFX, ICI, NCI, LSR, GSR, DFR, NIS, SIS
	SHRPOOL
MODCB EXLST	UPAD
MODCB RPL	TRANSID
RPL	TRANSID
SHOWCB ACB	BFRFND
	BUFRDS
	ENDRBA

Table 57. Unsupported Options of OS/VSAM Macros (continued)

OS/VSAM Macro	Unsupported Options
	HALCRBA
	LEVEL
	LOKEY
	NUIW
	RELEASE
	SHRPOOL
	UIW
SHOWCB RPL	TRANSID
TESTCB ACB	BUFRDS
	ENDRBA
	MACRF=CFX, NFX, ICI, NCI, LSR, GSR, DFR, NIS, SIS
	SHRPOOL
TESTCB RPL	TRANSID

With the following restrictions, OS format control blocks (ACB, RPL, EXLST) may reside above the 16MB line:

- The addresses contained in the ARG, ECB and MSGAREA fields of RPLs and PASSWD and MAREA fields of ACBs must be less than 16MB.
- The parameter lists for SHOWCB (ACB) and TESTCB (ACB) must be below the 16MB line.

OS/VSAM Error Codes

Error codes returned by VSE/VSAM in response to OPEN, CLOSE, and Data Management Request macro errors are mapped to the appropriate OS/VSAM error codes.

- Table 58 on page 476 lists the error codes returned by VSE/VSAM in response to OPEN errors.
- Table 59 on page 479 lists the error codes returned by VSE/VSAM in response to CLOSE errors.
- Table 60 on page 479 lists the error codes returned by VSE/VSAM in response to Data Management Request macro errors.

If a VSE/VSAM error code cannot be mapped to any OS/VSAM error code, a CMS error message and an ABEND 35 are issued except for the cases indicated by an "*".

The following table lists the VSE/VSAM to OS/VSAM error code mapping for OPEN errors:

Table 58. VSE/VSAM to OS/VSAM Error and Return Code Mapping for OPEN Errors

VSE/VSAM Error Code	CMS Error Message or OS/VSAM Error Code	VSE/VSAM Return Code	OS/VSAM Return Code
2	DMSVIP779E	8	N/A
4	4	8	8
14	DMSVIP782E	8	N/A
15	DMSVIP782E	8	N/A
17	DMSVIP782E	8	N/A

Table 58. VSE/VSAM to OS/VSAM Error and Return Code Mapping for OPEN Errors (continued)

VSE/VSAM Error Code	CMS Error Message or OS/VSAM Error Code	VSE/VSAM Return Code	OS/VSAM Return Code
18	DMSVIP782E	8	N/A
19	DMSVIP782E	8	N/A
32	DMSVIP782E	8	N/A
34	DMSVIP782E*	8	8
40	DMSVIP778E	8	N/A
48	168	8	8
50	DMSVIP782E	8	N/A
64	188	8	8
65	188	8	8
66	DMSVIP782E	8	N/A
67	DMSVIP782E	8	N/A
68	168	8	8
69	DMSVIP782E	8	N/A
70	DMSVIP782E	8	N/A
71	DMSVIP782E	8	N/A
72	148	8	8
78	DMSVIP782E	8	N/A
79	DMSVIP782E	8	N/A
80	DMSVIP778E	8	N/A
92	DMSVIP779E	8	N/A
96	96	4	4
100	100	4	4
104	104	4	4
108	108	4	4
110	160	8	8
113	144	0	4
114	DMSVIP781E	0	N/A
115	DMSVIP781E	8	N/A
116	116	4	4
117	DMSVIP782E	8	N/A
118	0	0	0
128	128	8	8
132	132	8	8
136	136	8	8

Table 58. VSE/VSAM to OS/VSAM Error and Return Code Mapping for OPEN Errors (continued)

VSE/VSAM Error Code	CMS Error Message or OS/VSAM Error Code	VSE/VSAM Return Code	OS/VSAM Return Code
144	144	8	8
148	148	8	8
152	152	8	8
160	160	8	8
161	160	8	8
162	96	4	4
163	132	8	8
165	DMSVIP782E	8	N/A
166	DMSVIP782E	8	N/A
167	DMSVIP782E	8	N/A
168	168	8	8
169	DMSVIP779E (This VSE/VSAM error code cannot be received when running OS/VSAM because CMS does not support LSR.)	8	N/A
180	180	8	8
188	DMSVIP782E	8	N/A
192	192	8	8
194	194	8	8
195	160	8	8
196	196	8	8
212	212	8	8
216	216	8	8
220	220	8	8
228	228	8	8
232	232	8	8
246	246	8	8
247	247	8	8
248	DMSVIP782E	8	N/A
254	DMSVIP782E	8	N/A
255	144	8	8

The following table lists the VSE/VSAM to OS/VSAM error code mapping for CLOSE errors:

Table 59. VSE/VSAM to OS/VSAM Error and Return Code Mapping for CLOSE Errors

VSE/VSAM Error Code	CMS Error Message or OS/VSAM Error Code	VSE/VSAM Return Code	OS/VSAM Return Code
2	DMSVIP783E	nonzero	N/A
4	4	nonzero	4
76	DMSVIP784	nonzero	N/A
136	136	nonzero	4
144	144	nonzero	4
165	DMSVIP784	nonzero	N/A
166	DMSVIP784	nonzero	N/A
167	DMSVIP784	nonzero	N/A
184	184	nonzero	4
188	0	nonzero	4
228	DMSVIP783	nonzero	N/A
246	246	nonzero	N/A
247	247	nonzero	N/A
252	DMSVIP784	nonzero	N/A
254	DMSVIP784	nonzero	N/A
255	148	nonzero	4

For Data Management Request errors, all VSE/VSAM error codes are returned to the OS/VSAM user because the VSE/VSAM and OS/VSAM error codes are equivalent, with the following exceptions:

Table 60. DATA Management Request Error Return Code Mapping

VSE/VSAM Error Code	CMS Error Message or OS/VSAM Error Code	VSE/VSAM Return Code	OS/VSAM Return Code
32	DMSVIP785E	0	N/A
48	40	8	8
52	Abend 34*	8	N/A
56	Abend 38*	8	N/A
128	DMSVIP786E	8	N/A
208	DMSVIP786E	8	N/A
212	DMSVIP786E	8	N/A
216	DMSVIP785E	8	N/A
224	48	8	8
229	229	8	8
245	245	8	8
246	246	8	8

Hardware Devices Supported

CMS support of VSAM data sets is based on VSE/VSAM. Therefore, only those disks supported by VSE/VSAM can be used for VSAM data sets in CMS.

These disks are:

- IBM 3390 Direct Access Storage
- IBM 9345 Direct Access Storage

CMS disk files used as input to or output from Access Method Services may reside on any disk supported by CMS.

Interface to an Alternate VSAM Emulator

CMS provides the support necessary to define and access an alternate VSAM emulator. Instead of using the VSE/VSAM support provided by CMS, you can write your own emulator or use some other VSAM emulator.

Note: Programs that intend to use an alternate VSAM emulator must be OS/VSAM programs.

To define an alternate VSAM emulator use the FILEDEF command. For example, if you have an emulator named FAST, you could use the following FILEDEF command:

```
FILEDEF VSAM SUBSYS FAST
```

The emulator (in this case, FAST) must already exist as a nucleus extension, be available as a module, or a be member of a loadlib. For more information on the SUBSYS option of FILEDEF, see *z/VM: CMS Commands and Utilities Reference*.

For the OPEN, CLOSE, and TCLOSE macros, the interface branches and returns by BALR to the origin of the alternate VSAM emulator module. On the call entry, the register contents are as follows:

R1
Standard subcommand PLIST address

R2
Address emulator nucleus extension SCBLOCK

R12
Entry point address

R13
24-word save area address

R14
Return address

R15
Entry point address

The parameter list pointed to by R1 may have one of two formats.

For OPEN requests:

	0	1	2	3	4	5	6	7
0	Name of the Alternate VSAM Emulator							
8	‘OPEN ‘							
16	Pointer to ACB				Reserved			
24	Emulator file name							
32	Emulator file type							
40	SYSPARM string length				Pointer to SYSPARM string			
48	X’FFFFFFFF’							

For CLOSE or TCLOSE requests:

	0	1	2	3	4	5	6	7
0	Name of the Alternate VSAM Emulator							
8	‘CLOSE ‘ or ‘TCLOSE ‘							
16	Pointer to ACB				Reserved			

The emulator's OPEN processing initializes the appropriate fields within the ACB according to the file processing macros (for example, GET, PUT, POINT, and so on).

The connection to the emulator is established as a result of issuing the FILEDEF command. The connection is an address pointer, at displacement 256 (X'100') within the OS/CVT simulated by CMS, to the origin of the emulator module. The address pointer is used by the control block manipulation macros (GENCB, MODCB, TESTCB, and so on) to establish the base address of the branch vector defined by OS/VSAM interface. The macros then branch directly to the appropriate branch table entry for the function being executed.

Appendix A. Sample Terminal Session for OS Programmers

"OSTEST ASSEMBLE" on page 483 shows an example assembler language program. "Commands to Execute OSTEST ASSEMBLE" on page 484 shows how you would assemble and execute it.

OSTEST ASSEMBLE

```
DATAPROC CSECT
         PRINT NOGEN
         SPACE
R0       EQU   0
R1       EQU   1
R2       EQU   2
R3       EQU   3
R10      EQU   10
R12      EQU   12
R13      EQU   13
R14      EQU   14
R15      EQU   15
         SPACE
         STM   R14,R12,12(R13)   save caller's regs
         BALR  R12,0               establish
         USING *,R12               addressability
         ST    R13,SAVEAREA+4 store addr of caller's savearea
         LA    R15,SAVEAREA   get the address of my savearea
         ST    R15,8(R13)     store addr in caller's savearea
         LR    R13,R15        save addr of my savearea
         SPACE
*open files and check that they opened okay
         SPACE
         LA    R3,0           initially set return code
         OPEN  (INDATA,,OUTDATA,(OUTPUT))     open files
         USING IHADCB,R10     get dsect to check files
         LA    R10,INDATA     prepare to check output file
         TM    DCBOFLGS,X'10' everything ok?
         BNZ   CHECKOUT       …continue
         LA    R3,100         set return code
         B     EXIT           …exit
CHECKOUT LA    R10,OUTDATA    check output file
         TM    DCBOFLGS,X'10' is it okay?
         BNZ   PROCESS        …
         LA    R3,200         set return code
         B     EXIT
         SPACE
PROCESS  EQU   *
         GET   INDATA         read a record from input file
         LR    R2,R1          save address of record
         PUT   OUTDATA,(2)    move it to output
         B     PROCESS        continue until end-of-file
         SPACE

EXIT     EQU   *
         CLOSE (INDATA,,OUTDATA)   close files
         L     R13,SAVEAREA+4 addr of caller's save area
         LR    R15,R3         load return code
         L     R14,12(R13)    get return address
         LM    R0,R12,20(R13) restore regs
         BR    R14            bye…
         SPACE
SAVEAREA DC    18F'0'
INDATA   DCB   DDNAME=INDD,MACRF=GL,DSORG=PS,         *
               RECFM=F,LRECL=80,EODAD=EXIT
OUTDATA  DCB   DDNAME=OUTDD,MACRF=PM,DSORG=PS
         DCBD
         SPACE
         END
```

"Commands to Execute OSTEST ASSEMBLE" on page 484 shows the sequence of commands that can be issued in order to assemble and execute the program in "OSTEST ASSEMBLE" on page 483.

Commands to Execute OSTEST ASSEMBLE

```
1   global maclib osmacro
    Ready;

2   assemble ostest
    ASSEMBLER DONE
    NO STATEMENTS FLAGGED IN THIS ASSEMBLY
    Ready;

3   filedef indd disk test data a
    Ready;

4   filedef outdd punch
    Ready;

5   cp spool punch to *
    Ready;

6   load ostest
    Ready;
    start
    DMSLIO740I Execution begins…

7   DMSSOP036E Open error code 04 on OUTDD.
    Ready(00200);
8   filedef
    INDD     DISK     TEST     DATA     A1
    OUTDD    PUNCH
    Ready;

9   filedef outdd punch (lrecl 80 recfm f
    Ready;

10  cp query reader all
    NO RDR FILES
    Ready;


11  load ostest (start
    DMSLIO740I Execution begins…

12  PUN FILE 6198  TO  BILBOCOPY 01 NOHOLD
    Ready;

13  fi indd reader
    Ready;
    fi outdd disk new osfile a4 (recfm fb block 1600 lrecl 80
    Ready;

14  listfile new osfile a4 (label
    DMSLST002E File not found.
    Ready(00028);

15  run ostest
    Execution begins…
    Ready;

6   listfile new osfile a4 (label
FILENAME FILETYPE FM FORMAT LRECL RECS BLOCKS DATE    TIME    LABEL
NEW      OSFILE   A4 F      1600  5    10     9/30/75 8:26:14 PAT198
Ready;
```

The following list explains the sequence of commands shown in "Commands to Execute OSTEST ASSEMBLE" on page 484.

1
: Since this assembler program uses OS macros, you must issue the GLOBAL command to identify the CMS macro library, OSMACRO MACLIB, before you can invoke the assembler.

2
: The ASSEMBLE command invokes the assembler to assemble the source file. The assembler completes without encountering any errors. If your ASSEMBLE file has errors, you should use the editor to correct them.

3
The FILEDEF command defines the I/O files used in this program. The ddnames INDD and OUTDD, defined in the DCBs in the program, must have a file definition in CMS. To execute this program, you should have a file on your A-disk named TEST DATA, which must have fixed-length, 80-character records. If you have no such file, you can make a copy of your ASSEMBLE file as follows:

```
copyfile ostest assemble a test data a
```

4
The output file is defined as a punch file so that it will be written to your virtual card punch.

5
The CP SPOOL command is issued, using the CP function, to spool your virtual punch to your virtual card reader.

6
The LOAD command loads the TEXT file produced by the assembly into virtual storage. The START command begins program execution.

7
An open error is encountered during program execution. The CMS ready message indicates a return code of 200, which is the value placed in it by your program.

8
The FILEDEF command, with no operands, results in a display of the current file definitions in effect.

9
Error code 4 on an open request means that no RECFM or LRECL information is available. An examination of the program listing would reveal that the DCB for OUTDD does not contain any information about the file format; you must supply it on the FILEDEF command. Re-enter the FILEDEF command.

10
You can use the CP QUERY command to determine whether there are any files in your card reader. It should be empty; if not, determine whether they might be files you need and, if so, read them into your virtual machine; otherwise, purge them.

11
Use the LOAD command to execute the program again; this time, use the START option of the LOAD command to begin the program execution.

12
The PUN FILE message indicates that a file has been transferred to your virtual card reader. The ready message indicates that your program executed successfully.

13
For the next execution of this program, you are going to read the file back out of your card reader and create a new CMS disk file in OS simulated data set format. FI is an acceptable system truncation for the command named FILEDEF.

14
The LISTFILE command is issued to check that the file NEW OSFILE does not exist.

15
The RUN command (which is an exec procedure) is used instead of the LOAD and START commands to load and execute the program. The ready message indicates that the program completed execution.

16
The LISTFILE command is issued again, and the file NEW OSFILE is listed. (If you issue another CP QUERY READER command, you will also see that the file is no longer in your card reader.)

Appendix B. Sample Terminal Session for DOS Programmers

Figure 65 on page 487 shows a sample assembler language program.

"Commands to Assemble and Execute DOSTEST ASSEMBLE" on page 487 shows how to assemble and execute the program.

Note: The assembler, in CMS, cannot read macros from VSE/AF libraries. This sample session shows how to copy macros from VSE/AF libraries and create CMS MACLIB files. Ordinarily, the macros you need should already be available in a system MACLIB file. You do not have to create a MACLIB each time you want to assemble a program.

```
BEGPGM   CSECT
         BALR  12,0
         USING *,12
         LA    13,SAVEAREA
         OPEN  INFILE,OUTFILE
LOOP     GET   INFILE
         PUT   OUTFILE
         B     LOOP
EODAD    EQU   *
         CLOSE INFILE,OUTFILE
          EOJ
          EJECT
BUFFER   DC    CL80' '
SAVEAREA DS    9D
INFILE   DTFDI MODNAME=SHRMOD,IOAREA1=BUFFER,DEVADDR=SYSIPT, *
               EOFADDR=EODAD,RECSIZE=80
OUTFILE  DTFDI MODNAME=SHRMOD,IOAREA1=BUFFER,DEVADDR=SYSPCH, *
               RECSIZE=81
SHRMOD   DIMOD TYPEFLE=OUTPUT
ENDPGM   EQU   *
         END
```

Figure 65. DOSTEST ASSEMBLE

Figure 66 on page 487 is an example of an ESERV file. You cannot have data in column 1 in an ESERV file. PUNCH is an ESERV program control statement that copies and de-edits macros from source statement libraries; in this case, the system source statement library. The output is directed to the SYSPCH device, which the CMS/DOS ESERV EXEC assigns by default to your A-disk.

```
    punch open,close,get,put,dimod,dtfdi
```

Figure 66. GETMACS ESERV, an ESERV file you need to assemble

"Commands to Assemble and Execute DOSTEST ASSEMBLE" on page 487 shows the sequence of commands issued to assemble and execute the program in Figure 65 on page 487.

Commands to Assemble and Execute DOSTEST ASSEMBLE

```
*
1   cp link dosres 130 130 rr linkdos
    DASD 130 LINKED R/O
    Ready;
    access 130 z
    Z (130) R/O - DOS
    Ready;
*
2   set dos on z
    DMSSET1101I 100K DOS partition defined at hexadecimal
    location 020000
    Ready;
```

```
*
3   assgn sysipt a
    Ready;
    eserv getmacs
    Ready;
*
4   listfile getmacs *
    GETMACS  ESERV    A1
    GETMACS  MACRO    A1
    GETMACS  LISTING  A1
    Ready;
*
5   maclib gen dosmac getmacs
    Ready;
    erase getmacs *
    Ready;
*
6   global maclib dosmac
    Ready;
*
7   assemble dostest
    ASSEMBLER DONE
    DOS00110        35           EOJ
    IFO078 UNDEFINED OP CODE
    NUMBER OF STATEMENTS FLAGGED IN THIS ASSEMBLY =     1
    Ready(00008);

*
8   listio sysipt
     SYSIPT  DISK      A
    Ready;
    eserv eoj
    Ready;
*
9   maclib add dosmac eoj
    Ready;
    maclib map dosmac (term
    MACRO    INDEX  SIZE
    OPEN         2    43
    CLOSE       46    43
    GET         90    56
    PUT        147    93
    DIMOD      241   647
    DTFDI      889   284
    EOJ       1174     6
    Ready;
*
10  erase eoj *
    Ready;
    assemble dostest
*
11  ASSEMBLER DONE
    NO STATEMENTS FLAGGED IN THIS ASSEMBLY
    Ready;
*

12  listfile dostest *
    DOSTEST  ASSEMBLE A1
    DOSTEST  LISTING  A1
    DOSTEST  TEXT     A1
    Ready;
    print dostest listing
    Ready;
*
13 doslked dostest *
    Ready;
*
14  listfile dostest *
    DOSTEST  ASSEMBLE A1
    DOSTEST  DOSLIB   A1
    DOSTEST  TEXT     A1
    DOSTEST  LISTING  A1
    DOSTEST  MAP      A5
    Ready;
*
15 cp spool punch to *
    Ready;
    punch test data a
    PUN FILE 0100  TO BILBO    COPY 01 NOHOLD
    Ready;
    cp query reader all
    Ready;
```

```
     ORIGINID FILE CLASS RECDS  CPY HOLD DATE  TIME
      NAME      TYPE    DIST
     PATTI    5840 A PUN 000097 01  NONE 09/29 15:00:39
      TEST      DATA    BIN211
*
16   assgn sysipt reader
     Ready;
     assgn syspch a
     Ready;
*
17   dlbl outfile a cms punch output (syspch
     Ready;
     state punch output a
     DMSSTT002E File not found.
     Ready(00028);
*

18   global doslib dostest
     Ready;
     fetch dostest
     DMSFET710I Phase DOSTEST entry point at location 020000.
     Ready;
*
19   start
     DMSLIO740I Execution begins…
     Ready;
     listfile punch output a (label
     FILENAME  FILETYPE  FM  FORMAT  LRECL  RECS BLOCKS
       DATE     TIME     LABEL
     PUNCH     OUTPUT    A1  F          80    97     10
       9/29/88 14:50:55  BBB191
     Ready;
     cp query reader all
     Ready;
     NO RDR FILES
*
20   assgn sysipt a
     Ready;
     dlbl infile a cms punch output (sysipt
     Ready;
     assgn syspch punch
     Ready;
*
21   fetch dostest (start
     DMSLIO740I Execution begins…
*
22   PUN FILE 5829  TO  BILBO    COPY 01 NOHOLD
     Ready;
     read punch2 output
     Ready;
     listfile punch2 output a (label
     FILENAME  FILETYPE  FM  FORMAT  LRECL  RECS BLOCKS
       DATE     TIME     LABEL
     PUNCH2    OUTPUT    A1  F          80    97     10
       9/29/88 14:50:59  BBB191
     Ready;
```

The following list explains the sequence of instructions shown in “Commands to Assemble and Execute DOSTEST ASSEMBLE” on page 487.

1

Use the CP LINK command to link to the DOS system residence volume and the ACCESS command to access it. In this example, the system residence is at virtual address 130 and is accessed as the Z-disk.

2

Enter the CMS/DOS environment specifying the mode letter at which the DOS/VS (VSE/AF) system residence is accessed.

3

You must assign the logical unit SYSIPT before you invoke the ESERV command. GETMACS is the file name of the ESERV file containing the ESERV control statements.

4

After the ESERV EXEC completes execution, you have three files. You may want to examine the LISTING file to check the ESERV program listing. The MACRO file contains the punch (SYSPCH) output.

5

The MACLIB command creates a macro library named DOSMAC MACLIB. Since the MACLIB command completed successfully, you can erase the files GETMACS ESERV, GETMACS LISTING and GETMACS MACRO; an asterisk in the file type field of the ERASE command indicates that all files with the file name of GETMACS should be erased.

6

Before you can invoke the assembler, you have to identify the macro library that contains the macros; use the GLOBAL command specifying DOSMAC MACLIB.

7

The ASSEMBLE command invokes the assembler to assemble the source file. The assembler displays errors encountered during assembly. The error listed indicates that the macro EOJ is not available since it was not copied from the source statement library. Create another ESERV file to punch this macro. The file EOJ ESERV contains one line:

```
     punch eoj
```

8

Use the LISTIO command to check that SYSIPT is still assigned to your A-disk so that you do not have to issue the ASSGN command again. Then issue the ESERV command again, this time specifying the file name EOJ.

9

Use the ADD function of the MACLIB command to add the macro EOJ to DOSMAC MACLIB. Then issue the MACLIB command again using the MAP function and the TERM option to display a list of the macros in the library.

10

Erase the EOJ files. You should always remember to erase files that you do not need any longer. Reassemble the program.

11

This time the assembler completes without encountering any errors. If your ASSEMBLE file still has errors, you should use the editor to correct them.

12

Use the LISTFILE command to check for DOSTEST files. The assembler created the files DOSTEST LISTING and DOSTEST TEXT. The TEXT file contains the object module. You can print the program listing if you want a printer copy. Then you may want to erase it.

13

Use the DOSLKED command to link-edit the TEXT file into an executable phase and write it into a DOSLIB. Because this program has no external references, you do not need to add any linkage editor control statements.

14

Now you have a DOSTEST DOSLIB containing the link-edited phase and a MAP file containing the linkage editor map. You can display the linkage editor map with the TYPE command or use the PRINT command if you want a printer copy.

15

To execute this program in CMS/DOS, punch a file that has fixed-length, 80-character records into your virtual card punch. If you do not have any files that have fixed-length, 80-character records, you can create a file named TEST DATA with the CMS Editor or by copying your ASSEMBLE source file with the COPYFILE command as follows:

```
copyfile dostest assemble a test data a
```

Use the CP SPOOL command to spool the punch to your own virtual machine, then use the PUNCH command to punch the file. The PUN FILE message indicates that the file is in your card reader. Use the CP QUERY command to check that it is the first or only file in your reader.

16

Use the ASSGN command to assign SYSIPT to your card reader and SYSPCH to your A-disk.

17

When you assign a logical unit to a disk mode, you must issue the DLBL command to identify the disk file to CMS. For this program execution, you are creating a CMS file named PUNCH OUTPUT. The STATE command ensures that the file does not already exist. If it does exist, rename it or else use another file name or file type on the DLBL command.

18

Use the GLOBAL command to identify the DOSLIB, DOSTEST if you want to search for executable phases; then issue the FETCH command specifying the phase name. The FETCH command loads the executable phase into storage. When the FETCH command is executed without the START option, a message is displayed indicating the entry point location of the program loaded.

19

The START command begins program execution. The CMS ready message indicates that your program completed successfully. You can check the input and output activity by using the LISTFILE command to list the file PUNCH OUTPUT. If you use the CP QUERY command, you can see that the file is no longer in your virtual card reader.

20

If you want to execute this program again, you can assign SYSIPT and SYSPCH to different devices; in this example, the input disk file PUNCH OUTPUT is written to the virtual punch. You do not need to reissue the GLOBAL DOSLIB command; it remains in effect until you reissue it or IPL CMS again.

21

This time the program execution starts immediately because the START option is specified on the FETCH command.

22

Again, the PUN FILE message indicates that a file has been received in your virtual card reader. You can use the CMS command READCARD to read it onto disk and assign it a file name and file type; in this example, PUNCH2 OUTPUT.

Appendix C. Sample Terminal Session Using Access Method Services

The sample terminal session in "Using Access Method Services under CMS" on page 493 shows you how to use access method services under CMS. You should have an understanding of VSAM and access method services before you use this terminal session.

The terminal session uses several CMS files, which you may create during the terminal session; or, you may prefer to create all of the files that you need beforehand. Within the sample terminal session, the file that you should create is displayed prior to the commands that use it.

This terminal session is for both CMS OS VSAM programmers and CMS/DOS VSAM programmers.

Note:

1. This terminal session assumes that you have, to begin with, a read/write CMS A-disk. This is the only disk required. Additional disks used in this exercise are temporary disks, formatted with the Device Support Facility program. If you have OS or DOS disks available, you should use them, and remember to supply the proper volume and virtual device number information, where appropriate. The number of cylinders available to users for temporary disk space varies among installations; if you cannot acquire ample disk space, see your system support personnel for assistance.
2. Output listings created by AMSERV take up disk space, so if your A-disk does not have a lot of space on it, you may want to erase the LISTING files created after each AMSERV step.
3. If any of the AMSERV commands that you execute during this sample terminal session issue a nonzero return code; for example:

   Ready(00012);

   You should edit the LISTING file to examine the access method services error messages. The publication*VSE/ESA Messages and Codes* contains the return codes and reason codes issued by access method services. You should determine the cause of the error, examine the DLBL commands and AMSERV files you used, correct any errors, and retry the command.

Using Access Method Services under CMS

```
1   cp define t3390 200 10
    Ready;
    DASD 200 DEFINED
    cp query virtual 200
    Ready;
    DASD 200 3390 (TEMP) R/W     10 CYL
*
    cp define t3390 300 10
    Ready;
    DASD 300 DEFINED
    cp query virtual 300
    Ready;
    DASD 300 3390 (TEMP) R/W     10 CYL
*
    cp define t3390 400 10
    Ready;
    DASD 400 DEFINED
    cp query virtual 400
    Ready;
    DASD 400 3390 (TEMP) R/W     10 CYL
*
2   type PUNCH DSF
     INIT UNIT(200) DEVTYP(3390) NVFY VOLID(222222)
       DVTOC(0,1,1) - MIMIC(MINI(10))
     INIT UNIT(300) DEVTYP(3390) NVFY VOLID(333333)
       DVTOC(0,1,1) - MIMIC(MINI(10))
     INIT UNIT(400) DEVTYP(3390) NVFY VOLID(444444)
       DVTOC(0,1,1) - MIMIC(MINI(10))
*
```

```
3   type DSF EXEC
     /* to Invoke Device Support Facility */
     arg cntrl .
     address command
     'CP CLOSE READER'
     'CP PURGE READER CLASS I'
     'CP SPOOL PUNCH CONT TO * CLASS I'
     'PUNCH IPL DSF S ( NOH'
     'PUNCH' cntrl 'DSF ( NOH'
     'CP SPOOL PUNCH NOCONT CLOSE'
     'CP SPOOL READER CLASS I NOHOLD'
     'CP IPL 00C CLEAR ATTN'
*

4   exec dsf punch
*
    NO FILES PURGED
    PUN FILE nnnn  TO  CAMPBEL COPY 001   NOHOLD
*
5   ICK005E DEFINE INPUT DEVICE, REPLY 'DDDD,VDEV' or
    'CONSOLE' ENTER INPUT/COMMAND:
*
6   2540,00c
    2540,00C
    ICK006E DEFINE OUTPUT DEVICE, REPLY 'DDDD,VDEV' or
    'CONSOLE' ENTER INPUT/COMMAND:
*
7   console
    CONSOLE
    ICKDSF -  SA DEVICE SUPPORT FACILITIES 5.0  TIME20:26:00
    03/09/82   PAGE  1
    INIT UNIT(200) DEVTYP(3390) NVFY VOLID(222222)
    DVTOC(0,1,1) - MIMIC(MINI(10))
    ICK00700I 200 BEING PROCESSED AS LOGICAL DEVICE = 3390
              PHYSICAL DEVICE = 3390-11
    ICK003D REPLY U TO ALTER VOLUME 200 CONTENTS, ELSE T
    ENTER INPUT/COMMAND:
*

8 u
  U
  ICK01314I VTOC IS LOCATED AT CCHH=X'0000 0001'
    AND IS     1 TRACKS.
  ICK00001I FUNCTION COMPLETED, HIGHEST CONDITION CODE WAS 0
*
   INIT UNIT(300) DEVTYP(3390) NVFY VOLID(333333)
    DVTOC(0,1,1) - MIMIC(MINI(10))
  ICK007001 300 BEING PROCESSED AS LOGOCAL DEVICE = 3390
     PHYSICAL DEVICE = 3390-11
  ICK003D REPLY U TO ALTER VOLUME 300 CONTENTS, ELSE T
  ENTER INPUT/COMMAND:
  u
  U
  ICK01314I VTOC IS LOCATED AT CCHH=X'0000 0001' AND IS
   1 TRACKS.
  ICK00001I FUNCTION COMPLETED, HIGHEST CONDITION
   CODE WAS 0
*
   INIT UNIT(400) DEVTYP(3390) NVFY VOLID(444444)
    DVTOC(0,1,1) - MIMIC(MINI(10))
  ICK00700I 400 BEING PROCESSED AS LOGICAL DEVICE = 3390
            PHYSICAL DEVICE = 3390-11
  ICK003D REPLY U TO ALTER VOLUME 400 CONTENTS, ELSE T
  ENTER INPUT/COMMAND:
  u
  U
  ICK01314I VTOC IS LOCATED AT CCHH=X'0000 0001' AND IS
   1 TRACKS.
  ICK00001I FUNCTION COMPLETED, HIGHEST CONDITION
   CODE WAS 0
*
  ICKDSF MAXIMUM STORAGE USED = 278968 BYTES
    (FIXED = 258120,
         DYNAMIC = 020848)
  ICK00002I ICKDSF PROCESSING COMPLETE. MAXIMUM CONDITION
    CODE WAS 0
*

9   cp ipl cms parm autocr
    Ready;
    CMS z/VM
    Ready;
```

```
*
10  cp link vseaf 350 350 rr pass=read
    DASD 350 LINKED R/O; R/W BY GANDALF
    access 350 z
    DMSACC723I Z (350) R/O - DOS
    Ready;
    set dos on z ( vsam
    DMSSET1101I 100K DOS partition defined at hexadecimal
    location 020000
    Ready;
*
11  access 200 b
    DMSACC723I B (200) R/W - OS
    Ready;
    access 300 c
    DMSACC723I C (300) R/W - OS
    Ready;
    access 400 d
    DMSACC723I D (400) R/W - OS
    Ready;
*
12  query search
    PLC191  191  A    R/W
    222222  200  B    R/W - OS
    333333  300  C    R/W - OS
    444444  400  D    R/W - OS
    MNT190  190  S    R/O
    MNT191  190  Y/S  R/O
    VSERES  350  Z    R/O - DOS
    Ready;
*
13  type MASTCAT AMSERV
     DEFINE MASTERCATALOG -
       (  NAME  (MASTCAT)  -
          VOLUME (222222) -
          CYL (4) -
          UPDATEPW (GAZELLE) -
          FILE (IJSYSCT) ) DATA (CYL(1))
*

14  assgn syscat b
    Ready;
    dlbl ijsysct b dsn mastcat (syscat perm extent
    DMSDLB331R Enter extent specifications:
    19 171
*
15

    Ready;
*
16  amserv mastcat
    Ready;
*
17  type CLUSTER AMSERV
     DEFINE CLUSTER ( NAME (BOOK.LIST  ) -
        VOLUMES (222222) -
        TRACKS (20) -
        KEYS (14,0) -
        RECORDSIZE (120,132) ) -
        DATA (NAME (BOOK.LIST.DATA) ) -
        INDEX (NAME (BOOK.LIST.INDEX ) )
*
18  amserv cluster
    4221A ATTEMPT 1 OF 2. ENTER PASSWORD FOR JOB AMSERV
      FILE MASTCAT
    gazelle
    Ready;
*
19  type REPRO AMSERV
      REPRO INFILE (BFILE -
            ENV ( RECORDFORMAT(F) -
            BLOCKSIZE(120) -
            PDEV (3390) ) ) -
            OUTFILE (BOOK)
*
20  assgn sys001 a
    Ready;
    copyfile test data a (recfm f lrecl 120
    Ready;
    sort test data a book file a
    DMSSRT604R Enter sort fields:
    1 14
    Ready;
    dlbl bfile a cms book file (sys001
    Ready;
*

21  assgn sys002 b
    Ready;
    dlbl book b dsn book.list (vsam sys002
    Ready;
    amserv repro
    Ready;
*
22  type SPACE AMSERV
      DEFINE SPACE -
           ( FILE (SPACE) -
             TRACKS (57) -
             VOLUME (333333) )
*
    assgn sys003 c
    Ready;
*
23  dlbl space c (extent sys003
    DMSDLB331R Enter extent specifications:
    19 57
*
    Ready;
*
24  amserv space
    4221A ATTEMPT 1 OF 2. ENTER PASSWORD FOR JOB AMSERV
      FILE MASTCAT
    gazelle
    Ready;
*
25  type UNIQUE AMSERV
     DEFINE  CLUSTER -
       ( NAME (UNIQUE.FILE) -
         UNIQUE ) -
       DATA  -
       ( CYL (3) -
         FILE (KDATA) -
         RECORDSIZE (100 132) -
         KEYS(12,0) -
         VOLUMES (333333 ) ) -
       INDEX -
       ( CYL  (1) -
         FILE (KINDEX) -
         VOLUMES (333333) )
*

26  dlbl kdata c (extent sys003
    DMSDLB331R Enter extent specifications:
    76 57
*
    Ready;
    dlbl kindex c (extent sys003
    DMSDLB331R Enter extent specifications:
    76 76
*
    Ready;
    amserv unique
    4221A ATTEMPT 1 OF 2. ENTER PASSWORD FOR JOB AMSERV
      FILE MASTCAT
    gazelle
    Ready;
*
27  type USERCAT AMSERV
     DEFINE USERCATALOG -
          ( CYL (8) -
            FILE (IJSYSUC) -
            NAME (PRIVATE.CATALOG) -
            VOLUME (444444) -
            UPDATEPW (UNICORN) -
            ATTEMPTS (2) ) -
           DATA  (CYL (3)  ) -
           INDEX ( CYL (1) ) -
           CATALOG (MASTCAT/GAZELLE )
*
28  assgn sys006 d
    Ready;
    dlbl ijsysuc d dsn private.catalog (extent sys006 perm
    DMSDLB331R Enter extent specifications:
    19 152
*
```

```
    Ready;
    amserv usercat
    Ready;
*
29  Tape 181 attached
*
30  type EXPORT AMSERV
      EXPORT BOOK.LIST  -
         INFILE (BOOK) -
         OUTFILE (TEMP  ENV (PDEV (2400)  REWIND NOLABEL ) )
*
31  dlbl book b dsn book list (cat ijsysct sys002
    Ready;
*

32  amserv export (tapout 181
    DMSAMS367R Enter tape output DDNAMEs:
    temp
    Ready;
*
33 type IMPORT AMSERV
    IMPORT  -
      CATALOG (PRIVATE.CATALOG/UNICORN) -
      INFILE (TEMP ENV (PDEV (2400) REWIND NOLABEL)) -
      OBJECTS (BOOK.LIST VOL (444444))
*
34  amserv import (tapin 181
    DMSAMS367R Enter tape input DDNAMEs:
    temp
    Ready;
```

The following list explains the sequence of commands shown in "Using Access Method Services under CMS" on page 493.

1
These commands define temporary 3390 minidisks at virtual addresses 200, 300 and 400.

2
This file contains control statements for the Device Support Facility program, which initializes disks for use by VSAM. These disks are labeled 222222, 333333 and 444444.

3
This file contains the commands necessary to use the Device Support Facility program in a virtual machine.

4
Execute the DSF EXEC, specifying that the Device Support Facility control statements contained in the file 'PUNCH DSF' should be appended to the stand alone Device Support Facility program.

5
These messages are issued by the Device Support Facility stand alone program.

6
Because the Device Support Facility control statements reside in the virtual card reader, you must indicate to Device Support Facility the device type and the address of your virtual reader.

7
This response tells Device Support Facility to output all run time information to your virtual machine console.

8
This response gives Device Support Facility permission to proceed with the initialization of the disk.

9
You must re-IPL CMS after all Device Support Facility processing has completed.

10
If you are a CMS/DOS user, you must access the VSE/AF SYSRES disk and issue the 'SET DOS ON fm (VSAM' command. If you have not previously linked to the VSE/AF SYSRES, you must use the CP LINK command before you issue the ACCESS command. Another method is to have the operator ATTACH the SYSRES disk to your virtual machine. Consult with your system programmer for the procedure to use at your installation.

11
ACCESS the three newly formatted disks as your B-, C- and D-disks. Note, the access modes you use need to be a letter before the letter R.

12
You can issue the QUERY SEARCH command to verify the status of all disks you currently have accessed. The 350 disk will be listed only if DOS is set on.

13
The file MASTCAT AMSERV defines the VSAM master catalog that you are going to use and provides space for suballocated clusters.

14
Identify the master catalog volume, and use the EXTENT option on the DLBL command so that you can enter the extents. For this extent, specify 171 tracks (9 cylinders) for the master catalog. Because 4 cylinders are specified in the AMSERV file, the remaining 5 cylinders will be used for suballocation by VSAM.

15
You must enter a null line to indicate that you have finished entering extent information.

16
Issue the AMSERV command specifying the MASTCAT file. The ready message indicates that the master catalog is created.

17
Define a suballocated cluster. This cluster is for a key-sequenced data set named BOOK.LIST.

18
No DLBL command is necessary when you define a suballocated cluster. Not that since the password was not provided in the AMSERV file, access method services prompts you to enter the password of the catalog, which is defined as GAZELLE.

19
Use the access method services REPRO command to copy a CMS data file into the cluster that you just defined.

20
You must identify the ddnames for the input and output files for the REPRO function. BFILE is a CMS file, which must be a fixed-length, 120-character file, and it must be sorted alphamerically in columns 1 through 14. The COPYFILE command can copy any existing file that you have to the proper record format; the SORT command sorts the records on the proper fields.

21
The output file is the VSAM cluster, so you must use the VSAM option on this DLBL command.

22
Create an AMSERV file to define additional space for the master catalog on the volume labeled 333333.

23
Again, use the EXTENT option on the DLBL command so that you can enter extent information and a null line to indicate that you have finished entering extents.

24
Issue the AMSERV command. Again, you are prompted to enter the password of the master catalog.

25
This AMSERV file defines a unique cluster, with data and index components.

26
You must enter DLBL command and extent information for both the data and index components of the unique cluster.

27
Next, define a private (user) catalog for the volume 444444. This catalog is named PRIVATE.CATALOG and has a password of UNICORN. Again, as in step 13, space is made available for suballocation.

28
When you define a user catalog that you are going to use as the job catalog for a terminal session, you should use the ddname IJSYSUC.

29
You may want to try an EXPORT/IMPORT function, if you can obtain a scratch tape from the operator. When the tape is attached to your virtual machine, you receive this message.

30
The file that is being exported is the cluster BOOK.LIST created above. If you do not have access to a tape, you can export the file to you CMS A-disk. Remember to change the PDEV parameter to reflect the appropriate device type.

31
You must reissue the DLBL for BOOK.LIST because there is a job catalog in effect, and the file is cataloged in the master catalog. Use the CAT option to override the job catalog.

32
There is no default tape value when you are using tapes with the AMSERV command. You must specify the TAPIN or TAPOUT option and indicate the virtual address of the tape. You are prompted to enter the ddname, which for this file is TEMP.

33
The last AMSERV file imports the cluster BOOK.LIST to the user catalog PRIVATE.CATALOG.

34
Read the tape in as input.

Appendix D. TSO Macros Simulated in CMS

CMS simulates a limited number of TSO macros. With the exception of the STAX macro, these macros are at an OS Release 20 level. The simulated TSO macros are provided primarily to support programs that were developed on TSO and ported to CMS.

The STAX macro is available in MVSXA MACLIB. All of the other TSO macros simulated by CMS are available in the OSMACRO MACLIB.

Although supported as a programming interface, the simulation of the TSO macros will not be upgraded (with the exception of STAX) and it is not recommended that you use them for application programming.

The following is a list of the TSO macros simulated by CMS:

GETLINE	IKJPOSIT	STATTN
GTSIZE	IKJPPL	STAUTOCP
IKJCPPL	IKJPSCB	STAUTOLN
IKJCSOA	IKJPTPB	STAX
IKJCSPL	IKJRLSA	STBREAK
IKJECT	IKJSTPB	STCC
IKJENDP	IKJSTPL	STCLEAR
IKJGTPB	IKJSUBF	STCOM
IKJIDENT	IKJTAIE	STSIZE
IKJIOPL	IKJUPT	STTIMEOU
IKJKEYWD	PUTGET	TCABEND
IKJLSD	PUTLINE	TCLEARQ
IKJNAME	RTAUTOPT	TGET
IKJPARM	SPAUTOPT	TPUT
IKJPGPB	STACK	TSABEND

For definitive information about these interfaces, including their programming classification (general use or product sensitive), please see the appropriate MVS TSO documentation.

Notices

Programming Interface Information

This document contains intended Programming Interfaces that allow the customer to write programs to obtain services of z/VM.

Trademarks

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.

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)
- 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).

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-Express Customer's Guide and Reference (https://www.ibm.com/support/pages/node/6019492), SA22-7935
- Open Systems Adapter-Express Integrated Console Controller User's Guide (https://www.ibm.com/support/pages/node/6019810), SC27-9003
- Open Systems Adapter-Express Integrated Console Controller 3215 Support (https://www.ibm.com/docs/en/SSLTBW_2.1.0/com.ibm.zos.v2r1.ioa/ioa.htm), SA23-2247
- Open Systems Adapter/Support Facility on the Hardware Management Console (https://www.ibm.com/docs/en/SSLTBW_2.1.0/com.ibm.zos.v2r1.ioa/ioa.htm), SC14-7580

Performance Toolkit for z/VM

- *z/VM: Performance Toolkit Guide*, SC24-6302
- *z/VM: Performance Toolkit Reference*, SC24-6303

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/servers/resourcelink/svc00100.nsf/pages/zosv2r5gc350033/$file/ickug00_v2r5.pdf), GC35-0033

Environmental Record Editing and Printing Program

- Environmental Record Editing and Printing Program (EREP): Reference (https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5gc350152/$file/ifc2000_v2r5.pdf), GC35-0152
- Environmental Record Editing and Printing Program (EREP): User's Guide (https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5gc350151/$file/ifc1000_v2r5.pdf), GC35-0151

Related Products

z/OS

- *Common Programming Interface Communications Reference (https://publibfp.dhe.ibm.com/epubs/pdf/c2643999.pdf)*, SC26-4399
- z/OS and z/VM: Hardware Configuration Definition Messages (https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc342668/$file/cbdm100_v2r5.pdf), SC34-2668
- z/OS and z/VM: Hardware Configuration Manager User's Guide (https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc342670/$file/eequ100_v2r5.pdf), SC34-2670
- z/OS: Network Job Entry (NJE) Formats and Protocols (https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sa320988/$file/hasa600_v2r5.pdf), SA32-0988
- z/OS: IBM Tivoli Directory Server Plug-in Reference for z/OS (https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sa760169/$file/glpa300_v2r5.pdf), SA76-0169
- z/OS: Language Environment Concepts Guide (https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sa380687/$file/ceea800_v2r5.pdf), SA38-0687
- z/OS: Language Environment Debugging Guide (https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5ga320908/$file/ceea100_v2r5.pdf), GA32-0908
- z/OS: Language Environment Programming Guide (https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sa380682/$file/ceea200_v2r5.pdf), SA38-0682
- z/OS: Language Environment Programming Reference (https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sa380683/$file/ceea300_v2r5.pdf), SA38-0683
- z/OS: Language Environment Runtime Messages (https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sa380686/$file/ceea900_v2r5.pdf), SA38-0686
- z/OS: Language Environment Writing Interlanguage Communication Applications (https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sa380684/$file/ceea400_v2r5.pdf), SA38-0684
- z/OS: MVS Program Management Advanced Facilities (https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sa231392/$file/ieab200_v2r5.pdf), SA23-1392
- z/OS: MVS Program Management User's Guide and Reference (https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sa231393/$file/ieab100_v2r5.pdf), SA23-1393

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

Index

Product Number: 5741-A09

Printed in USA

SC24-6257-73