z/VM
7.3

CMS Application Development Guide

SC24-6256-73

Note:

Before you use this information and the product it supports, read the information in “Notices” on page 625.

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: 2023-12-09

Contents

Figures

Tables

About This Document

The purpose of this document is to assist application programmers in the development of an application system to run on an IBM® z/VM® system. Application system means a group of applications (or just one application) needed to satisfy the given problem. This development process includes: planning and designing the application system, writing, compiling, and debugging an application using CMS services, and executing an application.

Intended Audience

This information is for experienced users in writing application programs in a high-level language or assembler language.

Before reading this information you should be familiar with the application development management cycle; that is, planning, designing, coding, debugging, and testing. You must also be familiar with the general concepts of CMS. If you are not familiar with CMS, see the *z/VM: CMS Primer*.

Where to Find More Information

For information about related publications, see the "Bibliography" on page 629.

Links to Other Documents and Websites

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

How to provide feedback to IBM

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

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

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.

SC24-6256-73, z/VM 7.3 (December 2023)

This edition includes terminology, maintenance, and editorial changes.

SC24-6256-73, z/VM 7.3 (September 2023)

This edition includes terminology, maintenance, and editorial changes.

SC24-6256-73, 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.

SC24-6256-02, z/VM 7.2 (September 2021)

This edition includes terminology, maintenance, and editorial changes.

SC24-6256-02, z/VM 7.2 (March 2021)

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

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

Information in the following topics is updated:

- “CMS Virtual Machine Environments” on page 4
- “Outline of VM Data Space Support” on page 218
- “Effect of Data Space Support on Existing Programs” on page 238
- “I/O Errors” on page 233

SC24-6256-01, 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.

SC24-6256-00, z/VM 7.1 (September 2018)

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

Part 1. Introduction

Before you begin to develop your application, you should be familiar with the environment in which you will be programming. The first chapter in this part describes the CMS environment and some of the programming services your application can use in CMS. The second chapter in this part describes OpenExtensions and the services that z/VM provides in support of open systems applications.

Chapter 1. Introduction to the CMS Programming Environment

z/VM is an operating system that lets you and many other users each have the functional equivalent of a computing system (the resources—processor, terminal, disk drives, tape drives, printers). This means that z/VM can support a large number of users who all need the machine at the same time.

Because you do not have direct control over the real machine, your configuration is known as a ***virtual machine***. Each virtual machine operates in the real computer under control of the control program (CP). The control program is the ***resource manager*** for the real computer. The resource manager ensures that each virtual machine is allocated the resources it needs to perform its own jobs.

Among the programs that can run in a virtual machine are the Conversational Monitor System (CMS) component of z/VM and operating systems such as z/OS®® and Linux® for z Systems®. This book discusses CMS.

What is CMS?

CMS provides functions for you to use at the terminal. It is an interactive environment. CMS is specifically designed to run on CP and depends on CP to interface with the resources. Therefore, unlike an operating system, CMS cannot operate independently on a real machine.

CMS provides terminal support, a file system, and a conversational command interface. (See Figure 1 on page 4.) CMS also allows you to run programs written in standard programming languages, such as COBOL, PL/I, VS Pascal, FORTRAN, C/C++, and assembler. CMS is designed to make the whole programming process easier. You can plan, design, code, run, and test an application in your own virtual machine without interfering with anyone else. It is similar to having your own workstation.

Although it seems like you are the only one using CMS, many other virtual machines are using CMS, and you or your application can communicate and share data with these virtual machines. You or your application can even communicate with a virtual machine on another VM system or with another user or application on a non-VM system. (See “Common Programming Interface (CPI) Communications” on page 7.)

Structure of CMS

CMS consists of three areas: the terminal support facility, the CMS system services, and the file system. Each area has a number of commands and programming interfaces that invoke CMS functions.

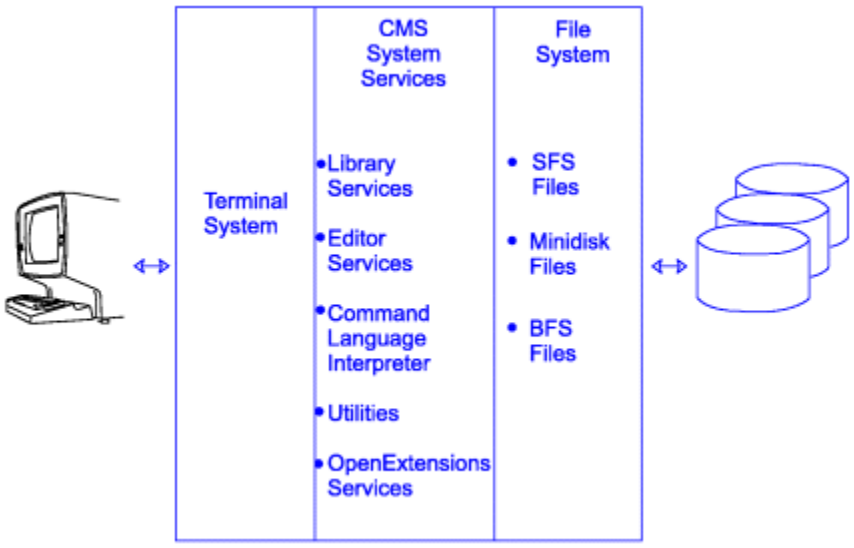


Figure 1. CMS System Structure

Terminal System
The CMS terminal system provides communication between you and your system. It reads commands entered at the keyboard and displays system responses to those commands.

File System
The CMS file system provides three methods for storing files. You can store files on virtual disks (called ***minidisks***), in ***Shared File System (SFS) directories***, or in ***Byte File System (BFS) directories***. SFS and BFS directories reside in CMS file pools.

Each file system provides basic input and output function, such as read and write operations. These I/O functions are used by the CMS system services and by applications running in a CMS virtual machine.

System Services
The CMS system services provide the basic user interface. It consists of:

- Library services
- Utility commands
- XEDIT
- REXX, EXEC 2, CMS EXEC
- The Batch Facility
- Message repository
- Resource recovery services
- Multitasking facilities
- OpenExtensions services (POSIX support)
- REXX sockets

CMS Virtual Machine Environments

To understand how your applications communicate with CMS and use available services, you must first understand the virtual machine environments in which CMS runs.

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, XA, or Z 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 instructions that operate on 64-bit registers. Existing ESA/390 architecture CMS programs can 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 runs 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 use 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 runs in z/XC mode.

Unless otherwise indicated, "CMS" means either version, and descriptions of CMS functions apply to both CMS and z/CMS. For more information, 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 that are 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, modules that are generated with the 370 option can run in an ESA/390 or ESA/XC virtual machine by issuing the CMS SET GEN370 OFF command. The 370 option of the GENMOD command is not supported,

Information about the 370 Accommodation Facility is available in another publication. See *z/VM: CP Programming Services*.

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

Information about the SET CMS370AC and SET GEN370 commands is available in another publication. See *z/VM: CMS Commands and Utilities Reference*.

The relationships between virtual machines and processor architectures are summarized in Table 1 on page 6.

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 architecture mode.
5. When z/CMS is IPLed in an ESA/390 virtual machine, z/CMS switches the virtual machine to z/Architecture and thereafter runs in z/Architecture mode.
6. Although z/CMS does not use or explicitly support 64-bit addressing mode, programs that run on z/CMS can enter 64-bit addressing mode.
7. Although z/CMS does not directly use storage above 2047 MB, z/CMS can be IPLed in a virtual machine with more than 2 GB of storage. z/CMS allows programs to use storage with addresses that are greater than 2 GB.
8. When z/CMS is IPLed in an XC virtual machine, z/CMS switches the virtual machine to z/XC and thereafter runs in z/XC mode.

CMS Programming Interface

The CMS programming interface enables applications using either 31-bit addressing or 24-bit addressing to run on CMS. The CMS programming interface consists of the following groups:

- CMS preferred interface group
- CMS compatibility group
- OS/MVS and DOS/VSE group.
- Systems Management APIs

CMS Preferred Interface Group

The CMS preferred interface group enables your application to be architecturally independent—your application can run in an ESA, XA, or XC virtual machine, and it can run on ESA/390 CMS or z/CMS.

The CMS preferred interface group consists of callable services (routines), macros, and functions that provide you with a means of making program calls, managing storage, performing I/O, handling interrupts, and processing abends. These routines, macros, and functions also 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 the ESA, XA, and XC virtual machine environments.

Applications written in the high-level languages use the routines. Applications written in the assembler language use the routines, the CMS macros, and the CMS functions.

All CMS routines are included in the preferred interface group. The routines 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 and file pool administration.
- *z/VM: CMS Application Multitasking* describes routines that perform multitasking and related programming tasks.
- *z/VM: OpenExtensions Callable Services Reference* describes routines that manage byte file systems, manipulate BFS data, and perform socket-related operations.

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

CMS Compatibility Group

The CMS compatibility group consists of 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, or XC virtual machines. Compatibility group interfaces may cause unpredictable results in 31-bit addressing mode.

See the *z/VM: CMS Macros and Functions Reference* for more information on the CMS compatibility group.

OS/MVS and DOS/VSE Group

The OS/MVS and DOS/VSE group consists of macros also provided by the OS/MVS and DOS/VSE operating systems. CMS supports these macros to make it easier to run CMS programs originally developed for OS/MVS or DOS/VSE.

See the *z/VM: CMS Application Development Guide for Assembler* for more information on the OS/MVS and DOS/VSE group.

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

Systems Management APIs

The Systems Management APIs provide a standard, platform-independent client interface that reduces the amount of VM-specific programming skills required to manage virtual system resources. These APIs include functions that create new virtual images, allocate and manage their resources, and change their configurations. They can be used to activate and deactivate images individually or in groups. Security and directory management functions are also provided.

The APIs are invoked by a client through a set of socket calls. See *z/VM: Systems Management Application Programming* for more information.

Common Programming Interface (CPI) Communications

Common Programming Interface (CPI) Communications defines a set of routines you can use to write ***Advanced Program-to-Program Communication (APPC)*** applications. You can call these routines from an application written in REXX, assembler or high-level programming languages. Programs using these routines can be more easily portable to other systems that abide by the CPI definitions.

z/VM has defined some routines that are extensions to CPI Communications routines. These z/VM extension routines exploit the capabilities of the z/VM operating system. In this book, the term "CPI Communications" includes the common routines and z/VM's extension routines.

CPI Communications lets your program communicate with another program that is on the same z/VM system, on a different VM system, or in a network defined by SNA. In z/VM, your application can use CPI

Communications only in a CMS environment. See Chapter 33, “Understanding CPI Communications,” on page 493 for more information on CPI Communications.

Resource Recovery Interface

The resource recovery element of the Common Programming Interface provides a consistent programming interface to the sync point management services. The sync point manager, through interactions with the underlying system support for various resources, provides services that allow changes to resources (such as database updates) to be made atomically—that is, either all are made (committed) or none are made (backed out). Affected resources can be either local or remote. See the *z/VM: CPI Communications User's Guide* for more information on CPI resource recovery.

The CMS Coordinated Resource Recovery (CRR) facility implements the CPI resource recovery interface in z/VM. CRR is available only in CMS environments. For more information, see Chapter 16, “Your Applications and Data Integrity,” on page 241.

REXX Sockets

z/VM supports an application program interface for socket applications written in REXX for the TCP/IP environment. The SOCKET external function in REXX/VM uses the TCP/IP IUCV interface to access the TCP/IP internet socket interface. This allows you to use REXX to implement and test TCP/IP applications. The REXX socket functions are similar to socket calls in the C programming language.

For descriptions of the REXX socket functions, see the *z/VM: REXX/VM Reference*.

CMS Operating Characteristics

CMS is a command-driven system—you enter a command, CMS runs 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 program (a program written by you, your local system programmer, or your favorite software house).

User-written programs are generally in the form of modules or execs. Modules are programs written in or compiled into machine-readable language. Execs are programs written in the Restructured Extended Executor (REXX) language. See the *z/VM: REXX/VM Reference* for more information on execs.

CMS Command Search Order

When you enter a command in the CMS environment, CMS uses a defined search order to locate the command. When the command is found, CMS stops its search and runs 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 run.

      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 with the file mode of the CMS installation saved segment. If the file mode of the saved segment is not closer to Z than the file mode of the directory or minidisk, then the exec on the saved segment is run. Otherwise, the exec in the directory or on the minidisk is run. 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 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 file type EXEC on any currently accessed disk or directory, using the standard 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" on page 8.
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 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 MODULE on any currently accessed disk or directory, using the standard 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" on page 9.

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 this 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 in the *z/VM: CMS Commands and Utilities Reference* for information on tables 1 through 4. See the SYNONYM command in the *z/VM: CMS Commands and Utilities Reference* for information on tables 5 and 6.

Preferred File Types

CMS has a list of preferred file types. The list of preferred file types are:

ASSEMBLE	FORTRAN	MODULE	TXTLIB
AUTOSAVE	HELP*	SCRIPT	XEDIT
CMSU*	LISTING	SYSU*	XEDTEMP
EXEC	MACLIB	TEXT	XMOD

Note: The "*" in the previous list represents the last four characters of the file type.

Preferred file types are used during the CMS search procedure to help eliminate unnecessary searches. Before searching an accessed disk for a file having a preferred file type, CMS determines whether the disk contains any files with the specified preferred file type. If CMS determines that the disk does not contain any files with the specified file type, CMS does not search the disk. CMS continues checking accessed disks until a disk is found containing files with the specified preferred file type or until all disks have been checked. This way, CMS only searches for the specified file on disks already determined to contain files with the specified file type.

Because this process improves performance by avoiding searching a disk for a file that is not located on that disk, you should keep groups of files with preferred file types on separate disks.

Programming Language Environments

z/VM supports many programming languages and applications. Some of the languages and applications supported and discussed throughout this manual are:

- Ada
- APL2*
- Assembler
- C
- C++
- COBOL
- FORTRAN
- Language Environment®
- MQSeries®
- Pascal
- PL/I
- REXX
- VisualAge® Generator

Note: If you want to use Java™ in a z/VM environment, consider using Java for Linux™ running in a Linux® guest.

The programming language environments are intended for anyone involved in planning or writing application programs. Application programmers use most of the programming language environments. Some languages, such as assembler, are of special interest to system programmers. Compiled high-level languages such as COBOL, PL/I, and FORTRAN use the same I/O interface on both MVS and z/VM. All compiled, high-level languages use the OS/MVS Simulation Interface when running on z/VM. If you run the same program on MVS and z/VM using the subset of the interface that OS/MVS Simulation supports, then the results should be equivalent. The most significant difference is that many conditions that result in abends in MVS are either ignored or simply give an error message in z/VM. See the *z/VM: CMS Application Development Guide for Assembler* for more information on OS/MVS Simulation.

The following documents provide information about programs that run on z/VM:

- *Licensed Products Migration Matrix for VM* lists IBM licensed programs. It is available at:

  IBM: z/VM Operating Systemrelated
- *Software Vendors' Products That Will Run on VM* lists non-IBM programs. It is available at:

  IBM: z/VM Operating Systemvendor

Another source of information about supported products is the IBM Global Solutions Directory (https://www.ibm.com/it-infrastructure/us-en/).

Chapter 2. Introduction to OpenExtensions

The term **OpenExtensions** refers to z/VM services that provide industry-standard open systems interfaces for applications and users, such as the Institute of Electrical and Electronics Engineers (IEEE) Portable Operating System Interface for Computer Environments (POSIX). This chapter provides an overview of OpenExtensions. It discusses compiling, building, and running applications that use OpenExtensions. It also includes a brief discussion of POSIX processes.

Overview

OpenExtensions is included in CMS and provides the z/VM implementation of the following POSIX standards:

- POSIX 1003.1 (known as POSIX.1) - System Interfaces
- POSIX 1003.1a (known as POSIX.1a) - Extensions to POSIX.1
- POSIX 1003.1c (known as POSIX.1c) - Threads
- POSIX 1003.2 (known as POSIX.2) - Shell and Utilities

Note:

1. Although OpenExtensions provides the `fork()` function for porting purposes, it does not meet full POSIX.1 specifications. (See the *z/VM: OpenExtensions POSIX Conformance Document* and the description of the fork (BPX1FRK) service in the *z/VM: OpenExtensions Callable Services Reference*. Also see "Converting fork() and exec() Usage to spawn()" on page 18.) As a substitute for `fork()`, OpenExtensions implements the `spawn()` function from POSIX.1d. All other parts of POSIX.1 are implemented at the Federal Information Processing Standard (FIPS) 151-2 level. For POSIX.1c, OpenExtensions provides a subset of the sixth draft of the threads standard.
2. Each of the POSIX.2 utilities additionally conform to the X/Open Portability Guide, issue 4 (XPG4) for Commands and Utilities.

The supported POSIX interfaces are provided as C/C++ library routines in the C/C++ run-time library included in Language Environment®. For programs written in other languages, a language-neutral version of the POSIX functions is provided as a set of OpenExtensions callable services library (CSL) routines. As shown in Figure 2 on page 11, these CSL routines are called by the C/C++ routines to provide the functions, but are also available to other applications. Programming language binding files are provided for REXX and Assembler (H or XL) application usage of these CSL routines. In addition, a REXX subcommand environment, ADDRESS OPENVM, is provided so the routines can be invoked as REXX functions.

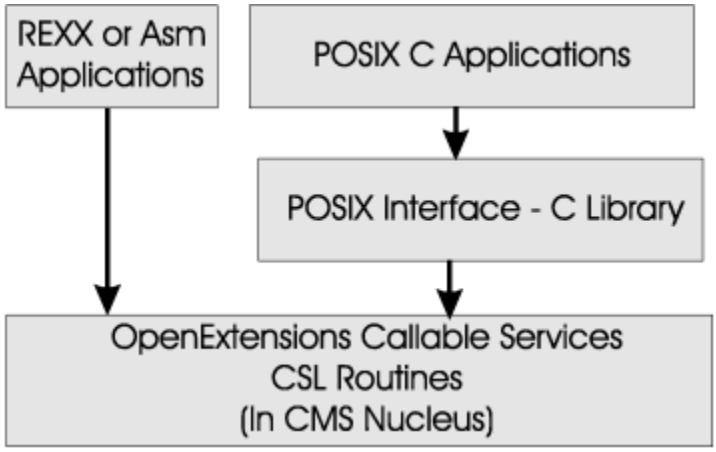


Figure 2. OpenExtensions Programming Interfaces

Included in OpenExtensions is a POSIX-compliant file system known as the byte file system (BFS). BFS is a companion to the CMS shared file system (SFS) that provides a byte-stream view of files rather than records. BFS allows data to be organized and used in a UNIX® style and format. Like SFS files, BFS files are organized in a hierarchical directory structure and stored in CMS file pools. While supporting the POSIX file system functions and rules, BFS also takes advantage of administration and system management facilities that it shares with SFS. These include space allocation, backup, and DFSMS/VM file migration, as well as other administrative functions.

CMS provides a set of commands, known as the OPENVM commands, that allow users to manage their BFS directories and files and control their related permission and ownership attributes. CMS Pipelines additionally provides the ability to use BFS from pipeline programs.

OpenExtensions provides a set of utilities that aid in program development and in porting applications from other open systems. OpenExtensions also provides a UNIX-like interactive user environment known as the shell. Users of the shell environment have access to both the shell command set (built-in commands and utilities) and the full CP and CMS command set, as well as both OpenExtensions and non-OpenExtensions applications.

Setting Up OpenExtensions

Before users can run OpenExtensions applications, you need to set up the OpenExtensions facilities in z/VM. This involves assigning POSIX security values to users and setting up the OpenExtensions byte file system. For more information, see the *z/VM: OpenExtensions User's Guide*.

OpenExtensions Byte File System

POSIX-compliant file services are provided by the OpenExtensions byte file system (BFS). CMS file system programming interfaces include both record-oriented services, provided by record file system assembler macros and callable services library (CSL) routines, and byte-oriented services, provided by OpenExtensions callable services.

The CMS record file system (which includes minidisks and SFS) supports record-oriented files with eight-byte file names and file types. In contrast, BFS supports byte-oriented files identified by a directory path and file name. CMS file pools serve as the repositories for both SFS data and BFS data. CMS provides a means for OpenExtensions applications to access CMS files stored on minidisks or in SFS, which allows new OpenExtensions applications to use traditional CMS production data.

The file paradigm used by BFS is significantly different from that of the CMS record file system. A BFS file is a byte stream whose interpretation is defined by the applications that use it. Such a file has no record format or other record attributes. BFS files are sometimes interpreted by special characters included in the byte sequence, such as the new-line character. Other differences from record files are sharing and update processing rules.

For the user, the chief differences between BFS files and record file system files are in the area of naming. A BFS file identifier includes a path and a file name. The path shows the file's position in the hierarchical directory structure, while the file name corresponds generally to a CMS file name. BFS has no concept of file type and file mode.

Figure 3 on page 13 provides an overview of file I/O interoperability between the record file system and BFS.

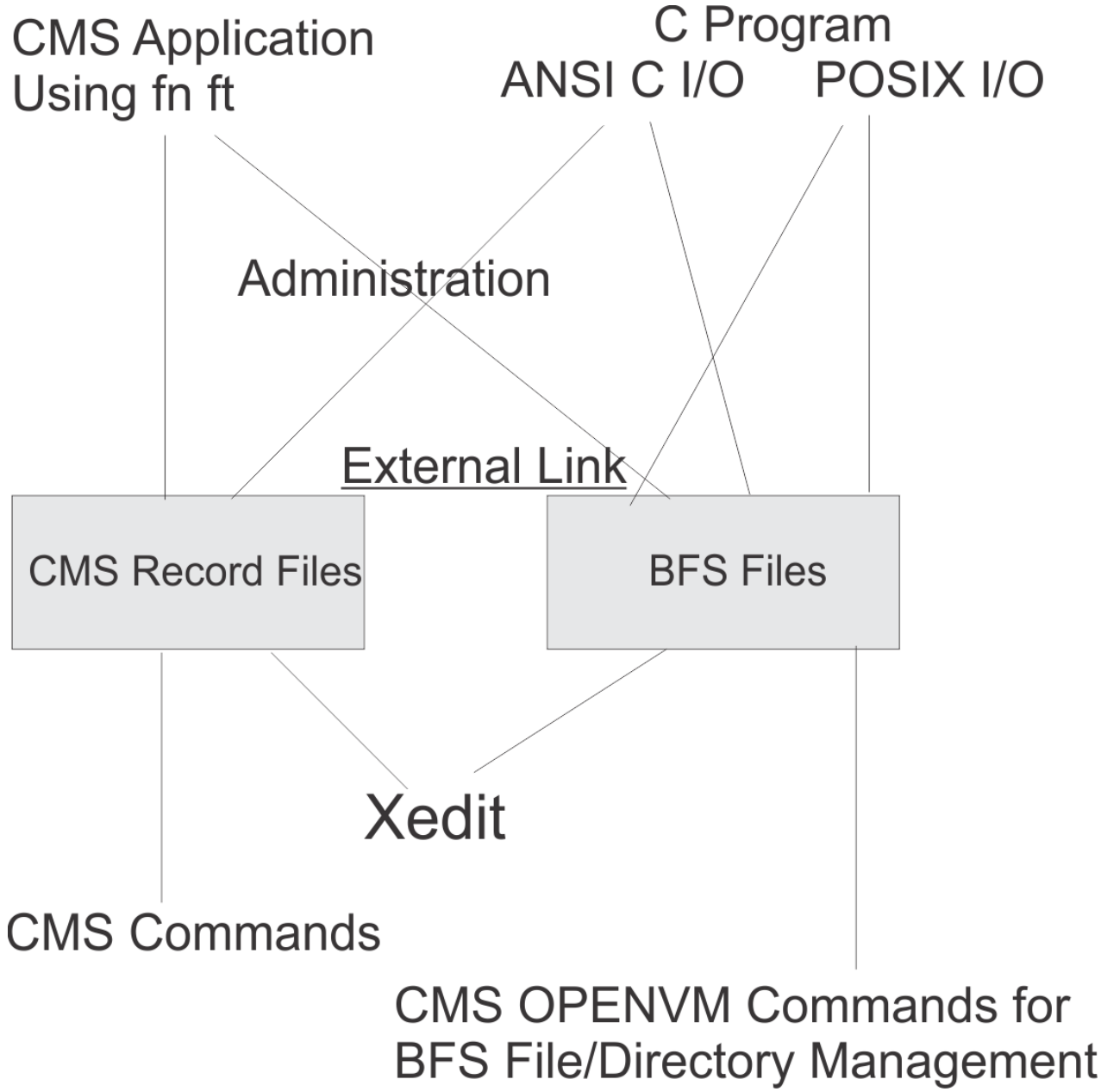


Figure 3. File I/O Interoperability

An OpenExtensions C/C++ application can access both record files and BFS files. The standard C/C++ file I/O functions, including `fopen()`, `fclose()`, `fread()` and `fwrite()`, can access minidisk files and SFS files through C/C++ naming convention and extensions that allow the programmer to specify file characteristics. The naming convention is simply to use a dot (.) between the file name, file type, and file mode elements of a CMS file ID and to begin the file ID with two slashes (//).

The POSIX file I/O functions in the C/C++ run-time library do not provide for the specification of file characteristics, such as logical record length or record format, as these are not concepts embodied in the POSIX file architecture.

To allow the POSIX file I/O functions to apply to CMS minidisk and SFS record-oriented data, BFS provides a new type of link known as an external link. An external link is a BFS file that represents a file stored outside of BFS. The external link is created by the CMS command OPENVM CREATE EXTLINK. The characteristics of the record-oriented file, including the value of any new-line character that should be associated with record boundaries, are specified on this command. The C/C++ run-time library uses this link information to operate on the record-oriented file.

The following points summarize the relationships between the record file system and BFS:

1. An application that performs file I/O through the ANSI C/C++ file I/O services, such as `fopen()`, `fread()`, and `fwrite()`, can manipulate files in either the record file system or BFS. This is done within the I/O semantics defined by C/C++.
2. Through POSIX file services, such as `open()`, `read()` and `write()`, an application can access BFS files or, through an external link, a record file.
3. The CMS record file API, such as the FSREAD and FSWRITE macros, or the DMSREAD and DMSWRITE CSL routines, can manipulate record files stored on minidisk or in SFS.
4. If a file is to be accessed from CMS applications using the record file API and also from OpenExtensions applications using the POSIX file services, the file must be stored in the record file system and accessed from BFS through an external link. When this external link is created, a user specifies how the record-to-byte mapping is to be made. In addition, CMS file sharing and commit rules are followed.
5. The CMS data block interface to SFS can also be used to read and write BFS files at the 4KB block level. The files will appear to have a record format of fixed and logical record length of 1.
6. The CMS file pool administration and DFSMS facilities and the administration CSL routines apply to both SFS and BFS.

CMS also provides commands specifically to manage BFS directories and files. These commands, known as the OPENVM commands, allow a user to perform such tasks as copying, erasing, renaming, and changing permissions and ownerships of BFS files and directories. In addition, a CMS command allows files to be copied into or out of BFS, with appropriate handling of special characters. The OPENVM commands are described in the *z/VM: OpenExtensions Commands Reference*, with additional information about their use in the *z/VM: OpenExtensions User's Guide*.

The CMS system editor, XEDIT, can be used to edit either record files or BFS files.

CMS Pipelines handles data in BFS files in the same way it handles data in SFS files and on CMS minidisk. Applications and tools that use CMS Pipelines to read and write SFS files will typically require very little change to handle BFS files.

Compiling and Building OpenExtensions Applications

A C/C++ program that makes use of POSIX services must be compiled and built using the C/C++ application development utilities included with OpenExtensions. These are the `c89` and `cxx` commands and the `make` utility. You also need the C/C++ compiler. Application source code in BFS files or CMS native files can be compiled to create output object files residing either in BFS or in the CMS native file system.

You may also want to refer to the following publications:

- *z/VM: OpenExtensions Commands Reference*
- *XL C/C++ for z/VM: User's Guide*

Using c89

The c89 command, by default, invokes the IBM IBM C for VM/ESA compiler to compile C application source code. The application source code can be compiled and built at one time or compiled and then bound at another time with other application source files or compiled objects.

The syntax of the c89 command is:

```
c89 [-options …] [file.c …] [file.a …] [file.o …] [-l libname ]
```

where:

options
are the c89 options described in the *z/VM: OpenExtensions Commands Reference*.

file.c
is the source file.

file.a
is the archive file.

file.o
is the object file.

libname
is the archive library.

You can use the c89 command from within the shell or directly from CMS. If you use c89, you must keep track of and maintain all the source and object files for the application program. However, you can take advantage of the make utility and create makefiles to maintain your OpenExtensions application source and object files automatically when you update individual modules. The make utility must be run from the shell. See "Using make" on page 16.

The c89 command has defaults for the name and placement of the executable file to be generated. The placement of the intermediate object file output depends on the location of the source file.

- If the C source file is a BFS file, the default executable file is `a.out`, the object file is *file*`.o`, and both are created in the BFS working directory.
- If the C source file is a CMS native file, the default executable file is *file* MODULE, the object file is *file* TEXT, and both are created as CMS native files on the CMS minidisk or SFS directory accessed as A. Because the CMS file ID is always converted to uppercase, you can specify it in lowercase or mixed case.

You can use the –o option on c89 to specify the name and placement of the executable file to be generated. For example:

- To compile C source file `usersource.c`, located in the BFS working directory, and build the executable file `mymod.out` in the `/app/bin` directory, specify:

  ```
  c89 -o /app/bin/mymod.out usersource.c
  ```

  The object file `usersource.o` will be placed in the BFS working directory.
- To compile the C source file `MAINBAL C`, located on the B disk, and build the executable file `mainbal.out` in the `/u/parker/myappls/bin` directory, specify:

  ```
  c89 -o /u/parker/myappls/bin/mainbal.out //mainbal.c.b
  ```

  The object file `MAINBAL TEXT` will be placed on the A disk.

To compile a C source file to produce only an object file, use the –c option. For example:

- To compile source file `usersource.c` to create the default object file `usersource.o` in the BFS working directory, specify:

  ```
  c89 -c usersource.c
  ```
- To compile source file `approg.c` to create an object file as a file on the A disk, specify:

  ```
  c89 -c //approg.c
  ```

By default, the c89 command invokes the C prelinker and the CMS LOAD and GENMOD commands. To use the Program Management binder (CMS BIND command) instead of the prelinker, LOAD, and GENMOD, specify the `–W b,b` option. For example:

```
c89 -W b,b,NOTERM file.c
```

You can also choose to have the c89 command invoke the IBM IBM C/C++ for z/VM compiler. To do this, you must first specify the C/C++ compiler module (CBXFINIT) on the _CNAME environment variable by issuing the following command:

```
globalv select cenv setlp_cname cbxfinit
```

The IBM C/C++ for z/VM compiler always uses the Program Management binder.

Using cxx

The `cxx` command invokes the IBM IBM C/C++ for z/VM compiler to compile a C or C++ application program and build an executable file in one step. The `cxx` command has the same syntax as the c89 command. A C++ source input file has the form *file*`.cpp` or *file*`.cxx`. If the C++ input file is a CMS native file, the file type must be CPP or CXX. The IBM C/C++ for z/VM compiler always uses the Program Management binder.

Use the `-o` option on `cxx` to specify the name and location of the executable file. For example:

- To compile and build a C application program source file to create the default executable file a`.out` in the BFS working directory, specify:

  ```
  cxx usersource.c
  ```

- To compile and build a C++ application source file to create the `mymod.out` executable file in the `/app/bin` directory, specify:

  ```
  cxx -o /app/bin/mymod.out usersource.cpp
  ```

- To compile and build several C application source files to create the `mymod.out` executable file in the `/app/bin` directory, specify:

  ```
  cxx -o /app/bin/mymod.out usersource.c ottrsrc.c //pwapp.c
  ```

- To compile and build a C++ application source file to create the `MYLOADMD MODULE` file on disk A, specify:

  ```
  cxx -o //myloadmd.module usersource.cpp
  ```

- To compile and build a C application source file with several previously compiled object files to create the executable file `zinfo` in the `/approg/lib` BFS directory, specify:

  ```
  cxx -o /approg/lib/zinfo usersource.c existobj.o //PWAPP.C
  ```

Using make

You can use the `make` utility in the OpenExtensions shell to control the parts of your OpenExtensions C/C++ application. The `make` utility calls the c89 command by default to compile and link the programs specified in the previously created makefile. However, when creating the makefile, you can specify the `cxx` command to compile and build C++ programs.

The `/etc/startup.mk` file contains the `make` default rules.

For example, if you have the file `/u/jake/appwrk/makefile.c` that contains the dependencies for OpenExtensions C application program `primappl`, and you make changes to the source file `subordpgm.c`, you can recompile the application by entering:

```
cd appwrk
make -f makefile.c
```

The result is the same as if you had entered:

```
c89 -O -o primappl ./appwrk/subordpgm.c
```

Note: The `make` utility requires that any application program source files to be "maintained" through use of makefiles must reside in BFS. To compile and build C/C++ source files that are in CMS native files, you must use the `c89` or `cxx` command directly.

See the *z/VM: OpenExtensions Commands Reference* for a description of the `make` utility. For a detailed discussion on how to create and use makefiles to manage application parts, see *z/VM: OpenExtensions Advanced Application Programming Tools*.

Running OpenExtensions Applications

Because an OpenExtensions application is simply a program that uses POSIX services, it need not be distinguished from any other CMS applications. Users can run OpenExtensions applications without entering the shell. How the application is invoked depends upon whether it resides in the CMS record file system or in BFS.

If the OpenExtensions application resides on an accessed minidisk or SFS directory, it has a CMS file ID in which the file name is the name of the application and the file type is MODULE. It is run, like any other CMS module file, by entering its name. If the application resides in BFS, it must be run by using the OPENVM RUN command. BFS directories are not searched during CMS command resolution, so no BFS files will ever be invoked as a result of entering a command at the CMS command line. In addition, applications in BFS have directory paths associated with them and names longer than the eight characters of a CMS command name.

If you want to run the file `application1.a` in the directory `/u/home/apps`, you would issue:

```
openvm run /u/home/apps/application1.a
```

CMS keeps track of the current working directory, so you could first change the working directory to the directory that contains your application and then avoid entering it on the OPENVM RUN command. For example,

```
openvm set dir /u/home/apps
openvm run application1.a
```

When running OpenExtensions programs, you should be aware of the following differences between how programs are started in the shell environment compared with how they are started directly from CMS:

- In CMS, the principle way to set environment variables is to set them in the CENV group of GLOBALV. The OpenExtensions C application will initialize its environment variables from this GLOBALV group. The shell has additional mechanisms for setting environment variables that can be interrogated by applications.

  If the OpenExtensions application is started through the OPENVM RUN command, the environment variables HOME, LOGNAME, PATH, and SHELL are initialized even if they are not defined in GLOBALV. HOME is set to the initial working directory field in the POSIX user database, or `/` by default. LOGNAME is set to the lowercase representation of the z/VM user ID. PATH defaults to `/bin:/usr/bin`. SHELL is set to the initial user program field in the POSIX user database, or `/bin/sh` by default.
- If an OpenExtensions application resides on an accessed minidisk or SFS directory, it can be invoked by name from the CMS command line. However, because it does not reside in BFS, it has no BFS permission settings. This means that all such OpenExtensions applications are executable, as if they had the **execute** permission set.
- To start an application in a strictly-conforming POSIX environment, you must start from the shell or by means of the OPENVM RUN command. An application started from the CMS command line is not guaranteed to have the environment variables HOME, LOGNAME, and PATH appropriately set. The user is responsible for giving values for these variables if the application has dependencies on them.
- An external link can be created in BFS that points to a CMS module in the record file system. This program can be run through the OPENVM RUN command, but care must be taken to assure that programs set up to be so invoked are capable of handling the POSIX entry conditions as defined by the `exec()` function.

- Applications started by using OPENVM RUN or the shell are automatically given the run-time option POSIX(ON). Applications invoked from the command line must be given the POSIX(ON) option explicitly. This is done either by coding a `#pragma runopts(POSIX(ON))` statement in the C application, or by passing the run-time option at invocation time. This is done by specifying the option after the command name and followed by a slash (/). The parameters passed to the command follow the slash. For example, to invoke the program `myappl` and pass it two parameters, `parm1` and `parm2`, the user would enter the following at the command line:

  ```
  myappl posix(on) / parm1 parm2
  ```

  Multiple run-time options can be specified. To invoke the program mentioned above and pass it an environment variable in addition to those defined by GLOBALV, the user would enter:

  ```
  myappl envar('OUTDIR=/tmp/out1') posix(on) / parm1 parm2
  ```

POSIX Processes

POSIX.1 defines a process as a program that is running. A process owns various resources, such as a signal environment and environment variables, and has associated with it various attributes, such as an effective user ID and group ID. Implicit in the definition of a POSIX process is the fact that it can run concurrently with other processes.

As part of its application multitasking support, CMS implements its own native processes, which are also programs that can run concurrently with each other. CMS represents a POSIX process as a CMS process that has the additional POSIX resources associated with it.

POSIX processes have local copies of their POSIX process environment, but native VM resources, such as the file system search order, nucleus extensions, subcommand environments, CP and CMS settings, and the virtual devices are shared by all POSIX processes in the CMS session.

A POSIX process is created when an OpenExtensions C/C++ program with the POSIX(ON) run-time option is run or when the `spawn()` function is called. The OpenExtensions application, together with the storage it has allocated, is known as the POSIX process image.

Converting fork() and exec() Usage to spawn()

The `spawn()` function provides a fast, low-overhead mechanism for creating a new POSIX process to run a new program. This is the typical usage of the POSIX.1 `fork()` function. OpenExtensions includes the POSIX.1d `spawn()` definition, which was included in the standard to handle the following operations in one function:

1. Create a new process.
2. Perform operations typically done in the new process to prepare to run a new program. This includes file descriptor mapping, changing process group membership, job control, and altering the signal handling environment.
3. Invoke the new program through `exec()`.

To convert an application from using `fork()` and `exec()` to using `spawn()`, the following steps should be followed:

1. Replace the call to `fork()` with a call to `spawn()`, using the program name and program parameters from the `exec()` call.
2. Delete the call to `exec()`.
3. Determine the other parameters to `spawn()` by examining the calls made between the `fork()` and the subsequent `exec()` to change the environment for the new program:
   - Calls to `dup2()` should be replaced by entries in the file descriptor array.
   - The mask value in any `sigmask()` calls should be used in the signal mask member of the inheritance structure.

- Signals whose actions are defaulted through `sigaction()` calls should be included in the sigdefault member of the inheritance structure.
- A call to `setpgid()` should be replaced by an entry in the process group member of the inheritance structure.
- A call to `tcsetpgrp` should be replaced by an entry in the inheritance structure.

For examples of using `spawn()` instead of `fork()` and `exec()` and a detailed explanation of factors to consider concerning the conversion, see Appendix M, “Converting fork() + exec() to spawn(),” on page 615.

POSIX Terminal Interactions

POSIX.1 defines terminal input and output in terms of file system I/O operations to character special files that represent the terminal. Generally, this character special file is `/dev/tty`. This file must be created by using the `mknode` shell command or the `mknode()` system call.

POSIX terminal I/O can be done only in canonical mode. This means that data is written to and read from the terminal in a one-line-at-a-time manner. Specifically, data is not read from the terminal until an attention key, such as the enter key, is pressed. Applications that operate in noncanonical mode, also known as character or raw mode, are not supported by the OpenExtensions terminal facilities.

Additional Considerations

A user virtual machine communicates with the BFS server virtual machine over Advanced Program to Program Communication (APPC) connections. Two APPC paths are used by each POSIX process during its execution, and each mounted file system also requires two paths. This means that the user should have a MAXCONN value in the user's CP directory entry of at least 70 and perhaps more if the user's directory tree is composed of elements managed by many file pools.

Part 2. Developing Your Program

This part discusses each task needed to develop your application. These tasks are contained in the following chapters:

Chapter 3. Planning and Designing Your Program

This chapter identifies and describes the objectives and the z/VM facilities you need to consider during the planning phase of z/VM application development.

You develop and use applications to solve problems and make certain tasks easier and faster to do—applications automate a process. Planning is the first phase in the application development process. You, as an application programmer, planner, or both, decide how to build an application package to execute successfully and efficiently. Once you determine the application's requirements, you plan an application package that satisfies these requirements. An application package can be a single application within one virtual machine or a group of applications located in many different virtual machines communicating with each other.

Note: Throughout this book, the terms "application", "application package", "application system", and "program" are used interchangeably.

Before planning your application, you must:

- Understand the function of the application (what problem you must solve).
- Understand the environment the application will run in (z/VM CMS programming environment). See Chapter 1, “Introduction to the CMS Programming Environment,” on page 3 for information on the CMS programming environment.

After you are through planning your application, you begin designing your application by expanding the objectives set by your plan.

Planning Objectives

During the early phases of application development, you can divide your planning and designing considerations into two areas:

- “CMS Environment Considerations” on page 23)
- “Application Processing Considerations” on page 29).

CMS Environment Considerations

The CMS environment considerations consist of:

- Identifying the system architecture
- Determining system resources
- Language considerations
- Data recovery/data integrity
- Using data spaces
- Types of processing
- Portability
- Tailoring
- Packaging your application
- Making your application available
- Supporting your application.

Identifying the System Architecture

As described in the Introduction, ESA, XA, and XC virtual machines give your application the capability of exploiting 31-bit addressing and residing above the 16 MB line in virtual storage (up to 2047 MB).

When you process your application, you have to specify this addressing and residency function using the AMODE and RMODE options. See Chapter 6, "Loading and Running Your Program," on page 47 for more information on using the AMODE and RMODE options.

Determining the Functional Level of CMS and CP

You may also want to know the functional level of CMS and CP of your system, especially if your application supports several releases of VM and depends on this information to make execution decisions. For example, it may use this information to determine the availability or behavior of certain commands.

The DMSQEFL CSL routine is the recommended method to obtain the CP and CMS functional levels. DMSQEFL returns codes that indicate both the product and level of CMS and CP.

Determining System Resources

System resources may be difficult to determine at the planning stage. Once you define the hardware and software requirements of the application, such as the database system or file system you want to use or any extra security features you may want to have, then you can determine:

- How much virtual storage you need to set up your application system.
- How much storage space your application requires.
- How much processor time your application requires. If your application requires a large amount of processor time, you may want to run it in a batch environment or send it to a disconnected virtual machine to run. This frees your terminal for other work.

See the appropriate books describing the resource requirements needed for each licensed program or feature. Begin by looking in the planning material, administration material, or both for each licensed program.

Language Considerations

The programming language you decide to use depends on the:

- Function of the application. Some programming languages are better suited for certain functions than other languages. For example, FORTRAN is used for scientific and engineering applications. COBOL is used for business and commercial applications.
- Language compiler that is available to you on your system.
- Standards used within your work area.
- Standards the customer mandates.

Data Recovery/Data Integrity

The first thing you must consider is how critical your data is and where it is located. Will loss of data (or the temporary loss of its use) disrupt business or endanger security or safety? Is the data located at your installation, on the same system, or is it on another system or distributed among systems in an SNA network?

The rule of thumb is: The more critical your data is, the more carefully and thoroughly you must prepare for recovery in the event of a failure. Although that sounds obvious, it is nevertheless crucial. The key word is **prepare**. If you plan for it carefully, it is less likely to get lost in the shuffle when writing the application begins.

CMS provides support for data integrity through ***Coordinated Resource Recovery (CRR)***. Resources that participate in CRR are called ***protected resources***. In a nutshell, this means that changes made to protected resources in the same work unit are all either committed or rolled back in unison to maintain consistency among the data.

Before starting the first project utilizing CRR, check with your system administrator to be certain that the CRR recovery server is available. You cannot have coordinated resource recovery without this server.

Note: An IBM-supplied CRR recovery server can optionally be installed during the z/VM installation process, or your own CRR recovery server can be generated in a postinstallation procedure.

Use Transaction Tags to Aid Problem Determination

The CRR recovery server and resource managers, such as Shared File System (SFS), each keep a special log to aid in recovering protected resources. Your application should provide CMS with meaningful information to store in the CRR log so that if a problem arises requiring operator intervention, the operator can determine what needs to be done. Although the likelihood of such a severe problem occurring is remote, you should nevertheless prepare for it by providing transaction-specific information that helps the operator or administrator make the proper decisions. You can use the Set Transaction Tag (DMSSETAG) and Get Work Unit ID (DMSGETWU) CSL routines to provide this essential information. For details on DMSSETAG and DMSGETWU, see "Setting Up to Ensure Data Integrity" on page 243 and the *z/VM: CMS Callable Services Reference*.

General Considerations

Generally, an application that resides on a single virtual machine and only writes to one resource (for example, a file pool) on a work unit need not be concerned with CRR or transaction tags. However, a resource manager may participate in CRR in such a way that a coordinated commit is always done.

In any case, many applications today need to perform complex transactions, often in a distributed environment. In support of these applications CRR automatically handles any necessary coordination of protected resources when an application:

- Writes to more than one resource on a work unit
- Accesses remote protected data
- Is distributed. (Part of the application runs on one virtual machine and part runs on one or more other virtual machines or systems. These parts are connected by protected conversations.)

Your application does not require any special actions to invoke the CRR facility. The only addition you should make to your application is to have it specify the appropriate transaction tags for CMS to store on the CRR recovery server log (and logs of participating resource managers) to help the operator determine what needs to be done in the event of a resynchronization or a situation requiring operator intervention. It is also a good idea to have your application save all data pertaining to a transaction if certain error codes are encountered. For more information, see "Designing Your Application for Data Integrity" on page 242.

Some Important Issues to Address

- **Are all resources able to participate in CRR?**

  Check the documentation of each resource to be accessed to determine if it participates in CRR. The SFS and protected conversations (those initiated with the SYNCLVL=SYNCPT option) in CMS participate in CRR. Note that the CMS minidisk file system and the OpenExtensions Byte File System do **not** participate in CRR.

- **What should you do if other resources you are using do not participate in CRR?**

  Changes made to data controlled by resources that do not participate in CRR cannot be guaranteed to be consistent with the data controlled by participating resources. The reason for this is that when updates are made to a nonparticipating resource in a work unit containing protected resources, CMS does not count the nonparticipating resources as a part of the work unit. Therefore, it is possible for the protected resources to be committed while the nonparticipating resource is left unchanged. You must issue a commit for that resource using the product-specific commit routine. To reduce problems, design your application to issue resource-specific commits **first** to make the changes permanent to resources not participating in CRR. Then, if that commit succeeds, issue a coordinated commit for the protected resources. If the commit fails, you can still roll back the work unit. It is also a good idea to journal (keep a chronological record of changes to the data) the part of the transaction that updates the nonparticipating resource.

- **Do all resources involved offer the same recovery features, such as backward recovery, forward recovery, and so on?**

Now that your applications can update multiple resources with integrity using CRR, you could have an application that updates both participating SFS file pools and participating non-SFS resources. SFS supplies the capabilities to back up data and then to restore it (see the *z/VM: CMS File Pool Planning, Administration, and Operation*). Assume that the non-SFS resources have the same capabilities. If a media failure occurs (such as a DASD failure), you can restore both the SFS resources and non-SFS resources to the backup level. The non-SFS resources may also have forward recovery, which restores the user data up to the time of the DASD failure. SFS, however, does not have forward recovery of user data, so the SFS resources could not be restored up to the time of DASD failure. This would result in the resources not being restored to the same level. Therefore, when you design applications that take advantage of CRR to update different resources in the same coordinated transaction, you should consider whether the different resources have forward recovery and the implications of one of the resources not having forward recovery.

The administration guide for each resource should indicate whether the particular resource participates in CRR and what type of recovery is provided. If the resource allows tailoring of the recovery level supported, check with the system administrator to determine exactly what recovery capabilities are installed. Also determine if the resource manager logs transaction tags. Your design for recovery procedures is limited by the lowest level provided by the resources you plan to access.

- **What can you do to limit effects of an application or system failure?**

  If a failure occurs after a synchronization point (or even during one), CRR should be able to maintain data consistency, but when processing can be resumed, it may be hard to tell which transactions were completed successfully before the failure. Therefore, you should consider having your application keep a log of the work it is doing. This could save considerable time and trouble whenever normal processing is disrupted by a failure.

- **What situations are not recoverable?**

  Operator or resource manager intervention can result in inconsistent data that is not recoverable. For example, the operator could issue a command to roll back changes to a file, but the resynchronization process results in the changes being committed to the other files involved in the same transaction. Resynchronization occurs when a commit or roll back cannot be completed due to a severe failure. Resynchronization completes the request at a later time, when contact is restored with all affected resources. See "How CRR Works" on page 241 for more information on resynchronization.

  Here are some things you can do to reduce the chances of this situation occurring:

  – Use the transaction tag to indicate the preferred action for your application in the event that operator intervention becomes necessary during resynchronization. This should be in line with the policies of the computer center where you run your application.
  – If your application must perform critical processing, you might also consider keeping a record of each critical transaction until it has been successfully committed or rolled back. This can aid in manual recovery when resynchronization fails or is interrupted.
  – Use the Set Synchronization Point Options (DMSSSPTO) CSL routine to set defaults for your application that reflect the policies of the computer center where you can run your application. Consistency helps to avoid confusion, especially in times of crisis.

- **Design your application with CRR in mind**

  There are several things to keep in mind when designing your application to access multiple resources:

  – Establish protected conversations by specifying SYNCLVL=SYNCPT.
  – Follow Common Programming Interface (CPI) Communications protocols for protected conversations.
  – Set up standards for the content of transaction tags and call the DMSSETAG (Set Transaction Tag) CSL routine at appropriate points in your application.
  – Find out how the error data for each resource is formatted so your program can use the DMSGETSP (Get Synchronization Point Errors) CSL routine to retrieve detailed error information.
  – Use the Set Synchronization Point Options (DMSSSPTO) CSL routine to set defaults for your application that reflect the policies of the computer center where your application will run.

Considerations for Multiuser Server Applications

If you write an application (like a server) to accept requests from multiple users, you will want to supply your own routine to replace the CRR Wait (DMSCWAIT) CSL routine that CRR uses to wait for asynchronous requests. The reason for this is that CRR does not exploit multitasking. Therefore, when your application calls DMSCWAIT, your virtual machine goes into a wait state until the event completes. If your application is serving several clients, it is unable to service requests from any other clients until completion of the event that triggered the call to CRR Wait. By supplying your own routine to wait on the request, your application can continue serving other clients. For information on the requirements for such a routine, see Chapter 17, "Writing a CRR Wait Routine for Multiuser Server Applications," on page 251.

Using Data Spaces

Data spaces are areas of storage that a program can create and access outside its own virtual machine storage. A program can use data spaces to store data for its own use and to share with other programs running in other virtual machines. Using data spaces in an application can improve overall apparent performance in these ways by:

- Reducing the need for IUCV or APPC/VM interaction between client and server virtual machines
- Replacing virtual I/O, such as DIAGNOSE I/O, with paging I/O (such as data space mapping services), which is more efficient
- Allowing data needed only temporarily to reside in storage rather than using DASD files for it.

See "Using Data Spaces in Your Applications" on page 221 for information on using data spaces.

Data space support includes sharing capabilities that apply to primary address spaces as well as data spaces. The term data space refers to a data-only address space. The term address space is more general, referring to any contiguous area of virtual storage that is addressable by a virtual machine. In general, address space is used in this book for situations that apply to both primary address spaces and data spaces.

Here are some things to keep in mind as you design data space support into your application:

- To create a data space, the application must be executing in an XC virtual machine.
- The virtual machine (user ID) in which the application is to run must be authorized to create and delete data spaces. In other words, it must have an XCONFIG ADDRSPACE directory statement.
- If a data space is to be shared among other users or applications, the virtual machine that owns the data space must be authorized to share by specifying the SHARE option on the XCONFIG ADDRSPACE statement.
- Virtual machines that need access to more than 62 address spaces must have a larger access list allocated with an XCONFIG ACCESSLIST directory statement. Although only XC virtual machines can create and directly manipulate data in a data space, ESA and XA virtual machines can copy data from data spaces and can share their primary address space.
- When you create a data space, request a large enough size to handle the needs of your application. You specify the data space size in units of 4KB pages. The amount of storage you specify when you create a data space is the maximum amount the system will allow you to use in that space.
- Data space management can be performed in either primary space or access register execution mode, but to manipulate data in a data space, the application must be running in access-register mode.

Types of Processing

Your application can run in the following processing environments:

- ***Online processing***

  You should use online processing if your application requires user interaction. An example of an online application is the record keeping application used at a video store.

- ***Batch processing***

You should use batch processing for applications that do not require user interaction or applications that use large amounts of processor time. An example of a batch application is payroll processing.

For information on using the CMS batch facility, see Chapter 23, "Using the Batch Facility," on page 357.

- ***Disconnected virtual machine processing***

  You can send your applications to a disconnected virtual machine to run. This frees up your terminal.

Portability

If you want your communications application to be portable from one system to another, you should use the SAA defined CPI Communications (also known as SAA communications interface) routines. z/VM supports these routines.

For more information on the CPI Communications routines, see Chapter 33, "Understanding CPI Communications," on page 493 and the *Common Programming Interface Communications Reference (https://publibfp.dhe.ibm.com/epubs/pdf/c2643999.pdf)*. For more information on designing a portable application, see the *SAA Writing Applications: A Design Guide*.

If you want your application to run on both MVS and VM, you should code it using OS/MVS Simulation. For more information on OS/MVS Simulation, see the *z/VM: CMS Application Development Guide for Assembler*.

If you want your application to run on VM and UNIX and other POSIX-compliant systems, you should code it as an OpenExtensions application using C POSIX services. For more information, see the C/C++ User's Guide.

Tailoring the System

You can tailor some areas of z/VM to enhance the operation of your application. You can do this by modifying the System Profile exec (SYSPROF EXEC). For more information on tailoring your system, contact your system administrator and see the *z/VM: CP Planning and Administration*.

Packaging Your Application

You can package your application as:

- An EXEC file
- A TEXT file
- A MODULE file
- An immediate command
- A subcommand
- A nucleus extension.

See Chapter 20, "Using Execs," on page 333 for information on using execs. See Chapter 6, "Loading and Running Your Program," on page 47 for information on creating TEXT files and MODULE files. See the *z/VM: CMS Application Development Guide for Assembler* for information on creating an immediate command, a subcommand, or a nucleus extension.

Making Your Application Available

Once your application package is complete, you will want to make it available to users. You can store these applications:

- On minidisks
- In SFS directories
- In saved segments
- In BFS directories.

For information on differences between minidisks, SFS, and BFS, see "Storing and Manipulating Data" on page 29.

See the *z/VM: CMS File Pool Planning, Administration, and Operation* for information on enrolling users in a file pool and the *z/VM: CMS User's Guide* for information on accessing a file pool and granting authority to users to access a file or an SFS directory. See Chapter 11, "Understanding the CMS File System," on page 119 and the *z/VM: CMS User's Guide* for information on the minidisk system. See Chapter 27, "Using Saved Segments," on page 419 for information on using saved segments. See Chapter 11, "Understanding the CMS File System," on page 119 and the *z/VM: OpenExtensions User's Guide* for information on BFS.

Supporting Your Application

You must be aware of who will be supporting your application.

- Will the support person be a programmer on site?
- If you are part of a distributed environment, will the support person be a programmer at another remote site or at the central site?
- Will you have the ***programmable operator facility*** running? *Programmable operator facility* enables automatic filtering and routing of messages from a specified virtual machine to a logical operator virtual machine in a local distributed or mixed environment. For details on *programmable operator facility* see the *z/VM: CP Planning and Administration*.
- Will you be using the ***Single Console Image Facility (SCIF)***? SCIF allows one user logged on to a single virtual machine to control multiple disconnected virtual machines. For details on SCIF, refer to the *z/VM: Virtual Machine Operation*.
- Will you be using ***NetView****? NetView automates network operations and network problem analysis. For more information on NetView, see *Network Program Products General Information* and the *Network Program Products Planning*.

Application Processing Considerations

The application processing considerations consist of:

- Storing and manipulating data
- Communicating with users and applications
- Controlling I/O
- Distributed processing
- System, data, and program security
- Debugging and testing your application package.

Storing and Manipulating Data

Most applications receive data, send data, or manipulate data. When you are planning an application package, you have to identify the application's data file and database requirements. z/VM provides you with the following data file system architectures and database system:

- Enhanced Disk Format (EDF) architecture (CMS minidisk files)
- Shared File System (SFS) architecture
- OpenExtensions Byte File System (BFS) architecture
- Structured Query Language/Data System (SQL/DS) database.

Enhanced Disk Format (EDF) Architecture

EDF, referred to as the minidisk system, stores files on ***minidisks***. A minidisk is a location of Direct Access Storage Device (DASD) space allocated to you without having to dedicate an entire DASD pack to you (unless all its space is needed). This disk space is allocated in contiguous cylinders or blocks. The space is yours whether you use it all or not. The following figure illustrates the EDF or *minidisk* file system.

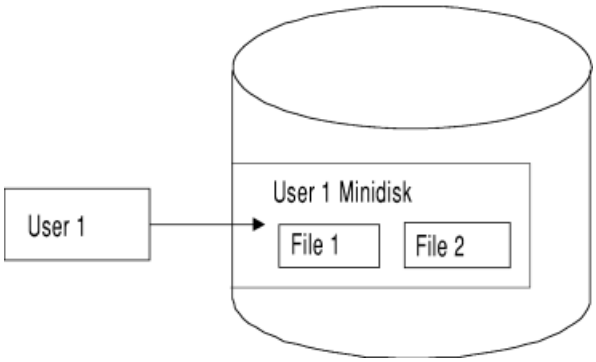


Figure 4. Minidisk System

Only one user can own a minidisk, but many users or applications can share data or other programs on a minidisk by linking to it. The following figure illustrates how users can share data or programs in EDF file system.

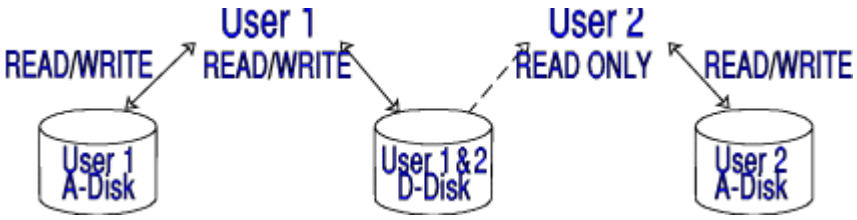


Figure 5. Users Sharing Disks in the Minidisk System

When you link to another minidisk, data access is at a disk level. That is, you have access to all the data files on this minidisk, except those files on the owners minidisk accessed as file mode A0. The owner of the minidisk cannot control who can look at the files.

Two users or applications should not attempt to write on a minidisk at the same time. There is no locking facility on the minidisk system. CMS does not protect a user from loss of data on a minidisk when multiple users have write access to it.

Shared File System (SFS) Architecture

Use SFS when you want to:

- Allow multiple users to share and update data and programs
- Allow users on different z/VM systems to access shared files
- Control multiple users from updating at the same time
- Ensure file level security (control who accesses specific files)
- Preserve data integrity. (SFS is a protected resource. It can participate in CRR.)

For list of performance considerations when you use SFS, see “Performance Tips” on page 184.

SFS stores files in ***CMS file pools***. A file pool is a collection of minidisks that contains the files for a number of users. This collection of minidisks is assigned to a virtual machine called a ***file pool server machine***. The file pool server machine manages all of the file within the file pool. An SFS user is given a certain amount of space within a file pool. This space is called a ***file space***.

File spaces are similar to minidisks because both are allocations of DASD storage for storing files. However, SFS stores files on different minidisks (transparently) within this file space. This space is not contiguous blocks of storage like the minidisk system. It is dynamic in that the space is allocated to you as you need it—up to the amount for which you have been authorized. This saves on DASD space because the whole space is not allocated and left unused. The following figure illustrates SFS.

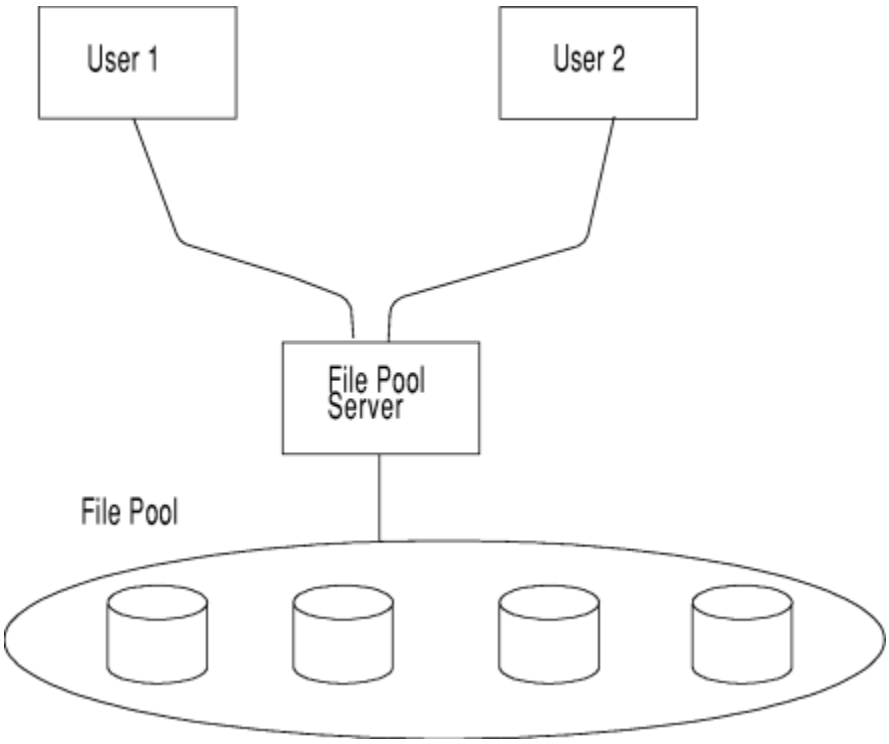


Figure 6. Shared File System

SFS lets you share files with other applications. You, as the owner of the files, can grant authorities that let others read from or write to your files and directories. The owner of the directory has control over who accesses the files.

Two applications cannot write to the same file within SFS at the same time. To prevent two applications from writing to the same file at the same time, SFS has a locking facility. There are two types of locks: an implicit lock and an explicit lock. SFS acquires and releases an implicit lock automatically. However, all implicit locks are released at the end of a ***work unit***. You can create an explicit lock by issuing a CMS command or routine that forces an object to be locked. An explicit lock can be in effect even if a work unit ends. A work unit is a group of related operations. SFS allows either all changes in a work unit to complete successfully or none of them to complete.

For more information on using SFS, see Chapter 12, “Manipulating SFS and Minidisk Files and Directories,” on page 129 and the *z/VM: CMS User's Guide*.

Manipulating Minidisk Files and SFS Files

The minidisk file system and SFS are collectively known as the CMS record file system, because data is stored in the form of records. You can manipulate CMS record files using a set of Callable Services Library (CSL) routines. A subset of these routines perform file I/O operations. These routines are easily used within your application.

For more information, see Chapter 12, “Manipulating SFS and Minidisk Files and Directories,” on page 129. You can also manipulate CMS record files using a group of CMS macros known as File System (FS) macros. See the *z/VM: CMS Application Development Guide for Assembler* for more information.

OpenExtensions Byte File System (BFS)

BFS is a hierarchical file system similar in concept to SFS. Like SFS files, BFS files are stored in CMS file pools. A byte file system is enrolled as a file space in a file pool. Multiple users can access the files and directories in a byte file system. 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. However, BFS files are different from CMS record files in that a BFS file consists of an ordered sequence of bytes rather than records. BFS allows data to be organized and used in a UNIX style and format.

A special set of CSL routines, known as the OpenExtensions callable services, are provided to allow applications to manipulate BFS files and directories. BFS data can also be accessed from CMS Pipeline programs. In addition, CMS supports limited manipulation of BFS files and directories using the CMS record file system interface (CSL routines and commands, but not FS macros).

For more information about BFS, see Chapter 11, "Understanding the CMS File System," on page 119 and Chapter 13, "Manipulating BFS Files and Directories Using CMS Record File System CSL Routines," on page 193. For information about the OpenExtensions CSL routines, see the *z/VM: OpenExtensions Callable Services Reference*.

DB2® Server for VM

DB2 Server for VM is a relational database management system. ***Relational*** means that the data is organized in table format. Each table has a certain number of columns and rows with data at each intersection. A ***database management system*** controls the storing and retrieval of this data.

Use a database system when you want to:

- Allow multiple users to share data and update data
- Control multiple users from updating at the same time
- Ensure row level security (control who accesses the data)
- Avoid inconsistencies
- Enforce data processing standards
- Access groupings of data with common characteristics
- Preserve data integrity.

DB2 Server for VM is a z/VM and SAA supported database system. DB2 Server for VM is a separately orderable licensed program. DB2 Server for VM simplifies data handling by offering facilities for querying data, manipulating data, and writing reports. It also contains data recovery and data security measures.

DB2 Server for VM allows you to run applications in either single user mode or multiple user mode. In multiple user mode, one or more users or applications can access the same database. DB2 Server for VM controls the access of data by allowing you to grant privileges to specific users on certain data or to give users access to only part of the information in a table. That is, you can control what rows in a table other users can access. Data access is at the row level.

To prevent two applications from updating the same information at the same time, DB2 Server for VM uses an automatic locking system. There are two types of locks: exclusive locks and share locks. Exclusive locks prevent other applications from reading or modifying data that your application is using. Share locks permit other applications to read data your application is using, but prevent other applications from modifying the data.

DB2 Server for VM preserves data integrity by using ***logical units of work***. A logical unit of work is a sequence of SQL statements that DB2 Server for VM treats as a single unit. Similar to the work units in SFS, either all changes in a logical unit of work complete successfully or none of them complete.

Your applications can use Structured Query Language (SQL) statements to access and manipulate data stored in the DB2 Server for VM database. (INSERT, DELETE, UPDATE)

For more details, see Chapter 28, "Using DB2 Server for VM," on page 425.

Communicating with Users and Applications

Communication is the primary reason for creating an application. An application can communicate with users interactively, with another application on the same system or on a different system, or with a database. You have to decide who your application communicates with and how it does this communicating.

Communicating with Users (Application to User Communication)

Your application can communicate with a user using a panel interface or a command interface.

A panel interface is usually much easier and friendlier for the user. When the application is panel driven, the user of the application does not have to be aware of how the information is processed. The user just enters data on a panel. If you use a command interface, the user of the application must know what commands are to perform the specific function and how to use these commands.

The following interactive facilities are available with z/VM:

- Interactive System Productivity Facility (ISPF)
- Display Management System for CMS (DMS/CMS).

For more details on the interactive facilities, see Chapter 24, "Creating an Interactive Program," on page 369.

Communicating with Applications (Application to Application Communication)

Your application can communicate with another application that runs in a virtual machine on the same system as your application, on a different z/VM system, or within a Systems Network Architecture (SNA) defined network. SNA is the IBM networking architecture. Applications communicate with each other using ***Advanced Program-to-Program Communication (APPC)***. The z/VM implementation of APPC is called APPC/VM.

z/VM provides two interfaces to APPC/VM: SAA defined CPI Communications routines and APPC/VM assembler functions. Your high level language applications can use the CPI Communications routines to communicate with applications on another z/VM system or on another operating system in the network. Assembler applications can use both the routines and macros to communicate with other applications.

For more details on application to application communication, see Chapter 30, "Introduction to Connectivity Programming in CMS," on page 459. For information on the individual CPI routines, see the *Common Programming Interface Communications Reference (https://publibfp.dhe.ibm.com/epubs/pdf/c2643999.pdf)*. For more details on application to application communication for assembler language applications, see the *z/VM: CMS Application Development Guide for Assembler*.

Controlling I/O

You should understand the routines and facilities that are available to control the input and output of your application. These I/O operations include:

- File I/O
- Screen and Terminal I/O
- Unit Record I/O
- Tape I/O.

File I/O

What language your application is written in and how you store your data determine how you perform file I/O operations. To access and manipulate files, your application can use:

- CSL routines
- Specific language statements
- FS macros
- CMS Pipelines stage commands: FILEFAST, FILEBACK, FILERAND, FILESLOW, <, >, >>, BLOCK, and DEBLOCK
- EXECIO command
- OS and DOS simulated macros.

For example, if your data files are stored in SFS, your application can use the CSL routines or FS macros to access and manipulate data. CSL routines can access data files stored in file pools and on minidisks. FS macros can also access data files stored in file pools and on minidisks. FS macros can be called only from an assembler program. Therefore, applications written in a high-level language must call assembler subroutines to use FS macros.

Note: FS macros can access only SFS data stored in file pools; they cannot access BFS data.

See the *z/VM: CMS Commands and Utilities Reference* for information on the EXECIO command. See the *z/VM: CMS Application Development Guide for Assembler* for information on the DOS and OS macros. See the *z/VM: CMS Pipelines User's Guide and Reference* for information on the CMS Pipelines stage commands.

Screen and Terminal I/O

Your applications can use the following methods to retrieve information from or display information on your screen or terminal:

- Dialog Management System—Interactive System Productivity Facility (ISPF) and Display Management System for CMS (DMS/CMS)
- XMITMSG command
- CMS Pipelines stage commands: FULLSCREEN, BUILDSCR, CONSOLE, FULLSCRQ, FULLSCRS, APLENCODE, APLDECODE, 3270ENC, 3270BFRA
- CONSOLE, LINERD, LINEWRT, APPLMSG macros.

Unit Record I/O

Unit record I/O operation consists of:

- Writing information to a virtual printer using the PRINTL macro or CMS Pipelines stage commands PRINTMC and URO
- Writing information to a virtual punch using the PUNCHC macro or CMS Pipelines stage command PUNCH
- Reading information from a virtual reader using the RDCARD macro or CMS Pipelines stage command READER

See Chapter 10, “Handling Input and Output,” on page 113 for more information on the I/O operations.

Distributed Processing

Distributed processing means that a specific *task* can be broken up into functions, and the functions are dispersed across two or more interconnected processors. A ***distributed application*** is an application for which the component application programs are distributed between two or more interconnected processors. ***Distributed data*** is data that is dispersed across two or more interconnected systems.

When your application is located on one system and the data you need is located on one or more other systems, you must decide whether you should write an application to access the distributed data or write a distributed application.

Use an application that accesses distributed data when:

- Your application runs in a batch environment. For example, if the function of your application only updates a database, this application can run in a batch environment.
- Your application does not access large amounts of data on another system.

However, accessing data that is distributed on another system is generally much less efficient than writing a distributed application because the data traffic increases. This may result in a decrease in the efficiency of the system.

Use a distributed application when:

- The data your application needs resides on another system, and your application expects this data to be processed before receiving it.
- Your application overwhelms the computing resources on one system. Then, you should divide the application into different functions, and let other systems do some of the processing.
- Your application performs a specific function that other applications can use. This application can reside on a system accessible by many applications.

- Your application is online.

Writing a distributed application is a more complex task than writing an application that accesses distributed data. When writing distributed applications, one application communicates with another application. These applications must be in sync with each other—that is, the application making a request must know when the application handling the request is ready to receive the request. In most situations, you will be writing both the requesting and the receiving applications. The two applications will use APPC to communicate.

Scenario 1

Suppose you have an application that accesses three files, all of which reside on one other system. If one subroutine of the program does all of the referencing of the three files on the other system, that subroutine could be moved over to the other system.

Scenario 2

Suppose you want to run an interactive application. You could divide this application into two sections: the panel presentation section and the logical (or main) section. In this example, the panel presentation processing could be done on another system and the logic processing could be done at the host system.

As a Rule of Thumb

If an application frequently accesses data, this application and data should reside on the same system.

System, Data, and Program Security

Security is another important element in your application environment. There are three areas of security to consider:

- System
- Data
- Program.

Before deciding what data and program security are needed for your application, you should understand your system security features and the security features of the products your application may be using. Customer requirements also determine the level of security needed in your program. Once you understand these security measures and customer requirements, you can determine the security measures you should use for your application.

System Security

To ensure that only authorized users access the z/VM system, the system administrator provides each user with a user ID and password. Only users with a valid user ID and password can access or log on to a z/VM system.

Data Security

If the data your application handles is sensitive, for example payroll information, it is critical to have control over who can access the data and how much of the data can be accessed.

Shared File System

SFS provides security features that control who can access data or programs. The file pool administrator controls the file pool. That is, the administrator defines the read and write authority on all SFS files owned by every user in a particular file pool. Each user can control who has access to the data or programs in their own directory. See Chapter 12, “Manipulating SFS and Minidisk Files and Directories,” on page 129 and the *z/VM: CMS File Pool Planning, Administration, and Operation* for details on the security features file pools provide.

Byte File System

BFS authorization mechanisms provide data security on an individual file basis. The system administrator provides each user of the system with a POSIX user ID (UID) and a primary POSIX group ID (GID). By default, if the user is not assigned a UID or GID, the system assumes a UID of 4,294,967,295 and a GID of 4,294,967,295. In addition, the user can be assigned membership in a number of supplementary groups.

When a user file space is created in a file pool, the UID of the root directory is derived from the *userid* parameter specified on the DMSENUSR routine (or the USER operand on the ENROLL command), and the GID of the root directory is derived from the *gname* parameter specified on DMSENUSR (or the GROUP operand on ENROLL). When a BFS file is created, the effective UID of the creating process and the GID of the parent directory become associated with the file.

There are three distinct classes associated with a BFS file:

file owner class
A process whose effective UID matches the UID associated with the file.

file group class
A process that is not in the file owner class, and whose effective GID or one of its supplementary GIDs matches the GID associated with the file.

file other class
A process that is not in either the file owner class or the file group class.

Each BFS file has its own set of permissions. These permissions are defined for each class (file owner, file group, and file other) and determine whether members of the corresponding class have read, write, or execute/search permission to the file. The owner of a BFS file can control access to the file by setting the permission bits associated with the three different classes. In addition, the file owner can change the file group of the file.

Database Management

A database management system, such as DB2 Server for VM, also provides security features that control who can access data. The database administrator controls the data and privileges associated with the database. A user must have connect authority to access the database. After receiving connect authority, a user can create tables and control who has access to the data in these tables.

DB2 Server for VM also allows different users to view different presentations of the same data. These features can prevent users from seeing data they should not see. See Chapter 28, “Using DB2 Server for VM,” on page 425 for details of the security features DB2 Server for VM provides.

Program Security

You may write certain applications that only a specific group of users can access. Using SFS, you can put your applications in certain directories and control who has access to these directories. Using BFS, you can set the permission bits of a file to control who has access to it.

If your application is handling requests from applications on other VM systems or non-VM systems, you may want to ensure that only authorized users communicate with your application. Therefore, as a local or global resource manager, your application should verify that each user attempting to communicate with your application is authorized to communicate.

If your application is managing a private resource, the system administrator can set up a special NAMES file, $SERVER$ NAMES, for the server virtual machine in which your private resource manager runs. z/VM uses the $SERVER$ NAMES file to check security authorization. The $SERVER$ NAMES file lists the private resources and the user IDs of the users authorized to access the private resources. Therefore, as a private resource manager, your application does not need to verify that users are authorized. See Chapter 32, “Program-to-Program Communications,” on page 479 for details on the $SERVER$ NAMES file.

If your application needs to communicate with another application, your application should specify access security information. The three types of access security your application can specify are: SECURITY(NONE), SECURITY(PGM), and SECURITY(SAME). You can use a CMS communications directory to define this access security information. A communications directory is a special NAMES file that lets APPC/VM applications connect to a resource using nicknames. See Chapter 32, “Program-to-Program

Communications," on page 479 for details on CMS communications directories and Chapter 31, "Understanding Communications Programming Terminology," on page 467 for details on the security types.

Debugging and Testing Your Application System

You can use CMS and CP to debug your applications. Most programming languages have a debugging tool you can use to find and correct errors in your application. You can also use the virtual machine environment z/VM provides to test your complete application package. For more information on the commands, debugging tools, and the "testing" environment, see Chapter 7, "Debugging and Testing Your Program," on page 65.

Chapter 4. Coding Your Program

In addition to the programming language statements you use when you write your application, z/VM also provides you with routines, macros, and functions your application can issue to perform a specific operation. This chapter briefly discusses the following services provided by z/VM:

- Callable services library (CSL) routines
- OpenExtensions callable services
- REXX Sockets
- CPI Communications (also known as SAA communications interface) routines
- CMS macros
- CMS functions
- DB2 Server for VM statements.

CSL Routines

VMLIB and VMMTLIB are CMS libraries that contain CSL routines. Rather than writing separate assembler routines, applications written in a high-level language (and also assembler applications) can use these routines to perform specific services. Some of the services provided by these routines are:

- Getting extended error information
- Extracting system information
- Opening and closing files
- Creating threads of execution for multitasking

See "Using Callable Services Libraries" on page 320.

Getting Extended Error Information

CMS maintains error blocks to store information regarding errors encountered by the various resources your applications access. Because the information may come from different sources and in different forms, CMS provides specialized CSL routines for retrieving it. The three routines provided are:

- DMSWUERR — Work Unit Error Data Deblocker
- DMSGETSP — Get Synchronization Point Errors
- DMSPCAER — Protected Conversation Adapter Errors.

All three routines are described in the *z/VM: CMS Callable Services Reference*.

DMSWUERR: DMSWUERR extracts the extended error information returned in the *wuerror* parameter on CSL routines that operate on the Shared File System (SFS) and places it in individual variables for high-level languages to use. The following information is returned in the *wuerror* buffer:

- Number of file pool error information areas returned
- Total number of errors for which information is available
- One or more groups of file pool error information.

The file pool error information includes the file pool ID, work unit ID, error reason code, and warning reason codes (up to 16 possible). The CSL routine, DMSWUERR, converts the *wuerror* output to data placed in individual variables. DMSWUERR can also be used to convert the SFS error information returned by the DMSGETSP routine.

Assembler programs can use the WUERROR and FPERROR macros to map into the workunit (WUERROR) and file pool (FPERROR) error data areas. These macros are described in the *z/VM: CMS Macros and Functions Reference*.

DMSGETSP: DMSGETSP retrieves all errors relating to the last synchronization point. This routine can return error blocks from a variety of resources, including SFS and the protected conversation adapter. Among the output parameters are *resource_component_ID* and *exit_name*. These two parameters can be used to identify what resource each error block is from so you can deblock the information correctly. For IBM products, the component ID can be found in the *Programming Systems General Information Manual*. You need to identify the product to determine the format of the error block, which is specified in the documentation of each product. For more information on using this routine, see the Chapter 16, "Your Applications and Data Integrity," on page 241.

DMSPCAER: DMSPCAER retrieves error information specific to a particular protected conversation. To use this routine, your program must first call DMSGETSP to get the CMS error block containing the information for DMSPCAER to parse. Because DMSGETSP retrieves all error blocks relating to the last sync point, your program must identify the error blocks containing the information you need by matching the resource component ID and exit ID of the protected conversation adapter for which you want error information. The DMSGETER (Get My Errors) CSL routine, described in the *z/VM: CMS Callable Services Reference*, can also be used to obtain the error blocks to be parsed.

Extracting System Information

Your applications can obtain or modify selected system information using the Extract/Replace facility. DMSERP is the CSL routine your application uses to call the Extract/Replace facility. For details on the Extract/Replace facility, see Chapter 14, "Extracting and Replacing System Information," on page 209. For details on the DMSERP routine, see the *z/VM: CMS Callable Services Reference*.

Opening and Closing Files

CMS provides CSL routines to open, close, and manipulate files residing in SFS and on minidisks. See Chapter 12, "Manipulating SFS and Minidisk Files and Directories," on page 129 for details on opening, closing, reading, and writing SFS and minidisk files.

OpenExtensions Callable Services

OpenExtensions interfaces, such as POSIX and socket functions, are provided as C/C++ library routines in the C/C++ run-time library included in Language Environment. For programs written in other languages, a language-neutral version of the functions is provided in CMS as a set of CSL routines (stored in VMMTLIB) known as the OpenExtensions callable services. These CSL routines are called by the C/C++ library routines to provide the functions, but are also available to other applications. Programming language binding files are provided for REXX and Assembler (H or XL) application usage of these CSL routines. In addition, a REXX subcommand environment, ADDRESS OPENVM, is provided so the routines can be invoked as REXX functions. These routines provide functions for manipulating Byte File System (BFS) files and performing socket-related operations. For more information, see the *z/VM: OpenExtensions Callable Services Reference*.

REXX Sockets

The REXX Sockets API provides support for socket applications written in REXX for the TCP/IP environment. The SOCKET external function of REXX/VM uses the TCP/IP IUCV API to access the TCP/IP internet socket interface. This allows you to use REXX to implement and test TCP/IP applications. The REXX socket functions are similar to socket calls in C.

REXX Sockets provides functions to:

- Process socket sets
- Initialize, change, and close socket sets
- Exchange data
- Resolve names for sockets
- Manage configurations, options, and modes for sockets

- Translate data and do tracing

For information about the REXX socket functions, see the *z/VM: REXX/VM Reference*.

CPI Communications Routines

In z/VM, APPC/VM enables applications to communicate with each other. CPI Communications is the "high-level language" interface to APPC/VM. CPI Communications consists of a group of routines that enable one application to communicate with another application. Applications written in REXX or a high-level language can use these routines.

See Part 4, “Connectivity Programming in CMS,” on page 457 for information on the CPI Communications routines. A complete description of each routine is found in the *Common Programming Interface Communications Reference (https://publibfp.dhe.ibm.com/epubs/pdf/c2643999.pdf)*.

Macros and Functions

CMS provides many assembler macros and functions. These macros and functions provide all the functions needed to write an application, ranging from writing lines to a terminal to abending the virtual machine.

See *z/VM: CMS Macros and Functions Reference* for details on the CMS macros and functions. See *z/VM: CMS Application Development Guide for Assembler* for information on how to use some of these macros and functions in your assembler applications.

DB2 Server for VM Statements

If your application needs data that is stored on an DB2 database, your application can issue DB2 commands to retrieve, insert, update, and delete this data.

See Chapter 28, “Using DB2 Server for VM,” on page 425 for details on coding and using DB2 commands.

Chapter 5. Compiling Your Program

This chapter includes information on:

- Invoking different language compilers
- Describing the files used and generated by the compilers
- Identifying libraries that contain files necessary to compile your application
- Identifying the location of your source files
- Specifying options on the compiler commands.

Before you run your application, you must compile your application. Compiling translates a program written in a high-level language into machine language. However, before you compile a program, you need to:

- Access the minidisk containing the compiler you need.
- Provide sufficient storage. See the individual language manual for storage requirements.
- Be aware of the file attributes of the individual source files. The file attributes are determined by the file type. See the *z/VM: XEDIT Commands and Macros Reference* for a list of file types and their file attributes.

Invoking the Compiler

Each language has a different command to invoke its compiler. The following table lists some of the languages z/VM supports and the corresponding command to invoke the language compiler.

Language	Compiler Command
VS COBOL II	COBOL2
OS PL/I	PLIOPT
VS Pascal	VSPASCAL
VS FORTRAN	FORTVS2
C/C++	c89 or cxx CC (for non-OpenExtensions program only)
Ada/370	A370 (for single source file) PKG370 (for multiple source files)

When you invoke these commands, CMS searches all of your accessed disks or directories, using the standard search order, until it locates the specified file. See "CMS Command Search Order" on page 8 for a description of the search order. The compiler creates an output listing file (file type of LISTING) and a text deck file (file type of TEXT). LISTING files contain the compilation list for each source file you compile. TEXT files contain the machine-language relocatable object code.

The compiler writes these files 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 the preceding choices are not available, the command is terminated.

Example: Suppose you have a FORTRAN program called TESTPROG FORTRAN. To compile the TESTPROG program using the VS FORTRAN Version 2 compiler, enter:

```
fortvs2 testprog
```

CMS creates two files on your read/write minidisk or directory: TESTPROG TEXT and TESTPROG LISTING.

The following figure shows the files the FORTRAN compiler uses:

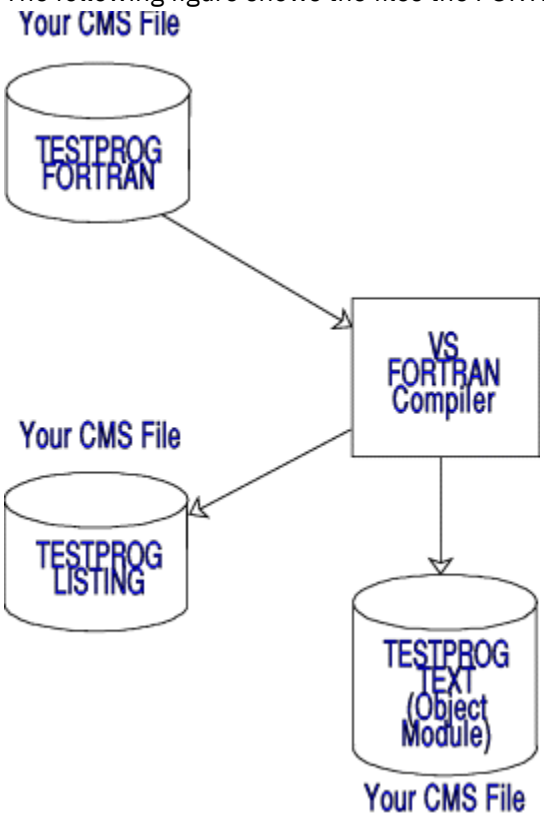


Figure 7. Files the FORTRAN Compiler Uses

If your FORTRAN program compiled correctly, you receive a message similar to the following:

```
VS FORTRAN VERSION 2 COMPILER ENTERED.  15:33:48
**TESTPROG** END OF COMPILATION 1 ******
VS FORTRAN VERSION 2 COMPILER EXITED.   15:33:48
Ready; T=0.06/0.02 15:33:49
```

If you had any errors in your program:

1. Edit the source program.
2. Correct the errors.
3. Compile the program again.

Identifying Source Files

Your source files can be located on a minidisk, in a directory, on tape, or in your virtual reader. If the source files are on a minidisk or in a directory, just invoke the compiler. If they are on tape or in your virtual reader, before invoking the compiler, you must do the following:

Source Files Located on Tape

If your FORTRAN source file is on tape, the source file must have been sent to tape using the MOVEFILE command. The MOVEFILE command puts the file in the correct format for the compiler. If the TAPE DUMP command is used, the file is not in the correct format and it will not compile correctly. To identify where the source file is located, enter the following FILEDEF command:

```
FILEDEF ddname TAPn (RECFM F LRECL 80 BLKSIZE 80
```

Ddname is the name the file is referred to in your program. Each language has a specific *ddname*. See the individual language manual for details. For example, if you were compiling a FORTRAN program, *ddname* would be FORTRAN. If you were compiling a PL/I program, *ddname* would be SYSIN.

The *n* is a number from 1 through 4 that corresponds to virtual tape units 181 through 184. `RECFM F LRECL 80 BLKSIZE 80` identifies the record format, the logical record length, and the block size.

Source Files Located in Your Virtual Reader

If the FORTRAN source file is located in your virtual reader, the source file must have been sent to the virtual reader using the PUNCH command with the NOHEADER option. The PUNCH command formats the file correctly for the compiler. The NOHEADER option is specified when you punch a file that is not going to be read by the RECEIVE command. If the source file was sent using the SENDFILE command, the file is not in the correct format. The application will not compile correctly.

You must also be aware of the file class of the source file. To ensure that you use the correct file when you invoke the compiler, you should change the class of the source file to a class not already assigned to the other files in your reader. (If the source file is the only file in your reader, you do not have to issue the CHANGE and SPOOL commands. You just have to issue the FILEDEF command.)

To change the class, enter the following CHANGE command:

```
CHANGE RDR spoolid CLASS c
```

spoolid is the spool ID number associated with the source file. *c* is the class you want the file changed to.

Then, you must make sure that your reader has the same class as the class of the source file. To do this, enter the following SPOOL command:

```
SPOOL READER CLASS c
```

c is the same class specified in the CHANGE command. See the *z/VM: CP Commands and Utilities Reference* for information on the CHANGE and SPOOL commands.

Next, to identify where the source file is located, enter the following FILEDEF command:

```
FILEDEF ddname READER (RECFM F LRECL 80 BLKSIZE 80
```

Ddname is the name by which the file is referred to in your program. Each language has a specific *ddname*. See the individual language manual for details. For example, if you were compiling a FORTRAN program, *ddname* would be FORTRAN.

Note: After a source file is compiled, it is erased from the reader.

Identifying Libraries to Be Searched

If you compile a source program that uses MACRO or COPY files, before you invoke the compiler, the macro libraries containing these files must be made available. Otherwise, CMS cannot find these files and you cannot compile your application successfully. To identify the macro libraries, issue the GLOBAL command.

For example, if COPYLIB MACLIB contains COPY files your application uses, enter the following GLOBAL command to identify the COPYLIB macro library:

```
GLOBAL MACLIB COPYLIB
```

You can specify more than one library on the GLOBAL command line. The libraries you specify on a GLOBAL command line are searched in the order you specify them. A GLOBAL command remains in effect for the remainder of your terminal session, until you issue another GLOBAL MACLIB command, or until you IPL CMS again.

See "Creating and Manipulating Macro Libraries" on page 306 for detailed information on creating macro libraries.

Specifying Compiler Options

When you compile your application, you may want to specify some options on the compiler command. The following list describes a few of these options. See the individual language manual for a complete list of their compiler options.

AMODE
specifies whether your application can run in 24-bit addressing mode or 31-bit addressing mode.

RMODE
specifies whether your application can reside above or below the 16 MB line.

XREF
produces a source cross reference listing that includes symbols and statement labels used in the source program.

Chapter 6. Loading and Running Your Program

This chapter includes information on:

- Defining your input and output files using the FILEDEF, CREATE NAMEDEF, and DLBL commands.
- Identifying libraries to resolve references using the GLOBAL command.
- Loading your application using the LOAD and INCLUDE commands.
- Running your application using the START, GENMOD, LKED, OSRUN, and OPENVM RUN commands.

Defining Input and Output Files

If your application has input or output files, before running your application, you must identify the files to CMS with the FILEDEF command. The FILEDEF command in CMS performs the same functions as the data definition (DD) card in MVS JCL: it describes the input and output files.

Similar to the CMS FILEDEF command is the CMS file system CREATE NAMEDEF command. The CREATE NAMEDEF command defines a namedef name that is to be associated with a real file, SFS directory, or file mode. This namedef name allows you to write application programs to use file system routines such that the file, directory names and file modes are defined externally to the program.

Using the FILEDEF Command

When you issue the FILEDEF command, you specify the following information:

- The **ddname**.
- The **device type**.
- A **file identifier**, if the device type is DISK.
- The **type of label** on your tape file, if tape label processing is specified.
- One or more **options**, as necessary.

The FILEDEF command connects the ***logical I/O control LIOCS (logical I/O control statements) statements (LIOCS)*** in your program with the ***physical I/O PIOCS (physical I/O statements) control statements (PIOCS)*** that define the I/O files outside the program.

Example: If you are writing a COBOL program, the SELECT or ASSIGN clause in the FILE CONTROL paragraph specifies the ddname. For example, your program could contain the following FILE CONTROL paragraph and FD statements:

```
FILE-CONTROL.
      SELECT INFILE ASSIGN TO UR-3505-S-TDATAIN.
      SELECT OUTFILE ASSIGN TO DA-3390-S-TDATAOUT.
       .
       .
       .
FD    INFILE
       .
       .
       .
FD    OUTFILE
       .
       .
       .
```

In this case, *ddname* for the input file is TDATAIN, and *ddname* for the output file is TDATAOUT. These are the names you would use in the *ddname* portion of the FILEDEF command. For example, if the input file is to be read from your virtual reader, enter the following FILEDEF command:

```
FILEDEF TDATAIN READER
```

Specifying Device Type

For input files, the device type you enter on the FILEDEF command line identifies the device from which you want the records read. The device type can be:

DISK
for files on CMS (minidisk or directory), OS, or VSE disks.

TERMINAL
for keyboard input.

READER
for input from your virtual reader.

TAP*n*
for tape. The *n* designates multiple tape drives.

For output files, the device you specify can be:

DISK
for files on CMS (minidisk or directory), OS, or VSE disks.

TERMINAL
for terminal output.

PRINTER
for the virtual printer.

TAP*n*
for tape. The *n* designates multiple tape drives.

PUNCH
for the virtual punch.

You may specify a FILEDEF, but may not need the output. For example, when you test a program, you may be concerned with its logic rather than the correctness of the output. In this case, you can specify the device type DUMMY. FILEDEF provides the necessary linkage between LIOCS and PIOCS, but no actual data is produced. Also, if your program can run without input, you can use the device type DUMMY when defining input files.

Specifying a CMS File ID for Input and Output

If the device type is DISK and the input or output file is on a CMS disk, you can specify a CMS file identifier as part of the device type specification.

In the preceding example, the TESTPROG COBOL program has an input *ddname* of TDATAIN. Suppose this file resides on your A-disk in a file called TEST DATA A1. To identify the file to CMS prior to running your program, enter the following command (on the command line or from within an EXEC):

```
filedef tdatain disk test data
```

Specifying FILEDEF Options

The FILEDEF command provides several options to control file specifications. Some of these options are:

BLOCK (or BLKSIZE), LRECL, RECFM, and DSORG
specifies and describes the file format and organization.

PERM
specifies whether the file definition is to be permanent (until CHANGE, or IPL (next session)).

MEMBER
specifies if the file is a member of an OS partitioned data set or CMS MACLIB or TXTLIB.

UPCASE/LOWCASE
determines whether output is in uppercase or mixed case.

See the *z/VM: CMS Commands and Utilities Reference* for a complete description of each option.

Identifying VSAM Files Using the DLBL Command

If your application uses VSAM files for input or output, use the DLBL command to identify these VSAM files to CMS. For details on the DLBL command, see the *z/VM: CMS Commands and Utilities Reference*.

Using the CREATE NAMEDEF Command

Specifically, a namedef is a 1- to 16-character string that represents one of the following:

- File name and file type
- Directory ID
- File mode letter

To associate a namedef with the name of a real file or directory, you issue the CREATE NAMEDEF command prior to running the program. When the program runs, CMS does the operations on the file or directory the namedef represents. Now by using namedefs, you can run a program to process different files, directories, or file mode letters without changing the code and recompiling the program. For more information on using namedefs, see "Using a Namedef" on page 131. For more information on the CREATE NAMEDEF command, see the *z/VM: CMS Commands and Utilities Reference*.

Loading Your Application

The compiler generates an object file (TEXT file) that contains machine-language object code. The TEXT file is also referred to as an ***object module***. To load a TEXT file into storage, issue the LOAD command, specifying the name of a TEXT file you want to load. To load subsequent TEXT files, use the INCLUDE command.

The INCLUDE command has a similar format and option list as the LOAD command. The main difference between them 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 operate, they produce a load map. You may find the load map useful in debugging your programs. The load map indicates entry points loaded, their virtual storage locations, and their AMODE and RMODE values. The load map shows what the default values are or what you may code in your application. The load map does not show what options you may have specified on the LOAD command.

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.

Where Are TEXT Files Loaded?

The relocatable object code references addresses relative to the start of an entry point. The reference point is always taken as zero. However, when you load your object module, the LOAD command loads the object module into storage beginning at X'20000' or the largest contiguous storage location available, unless otherwise specified. When the program is loaded, all address references within the module are resolved relative to the load point.

For example, if an object module references an address at X'30A' in the relocatable (TEXT) version, after issuing the LOAD command, all references to that address are changed to X'2030A'.

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 a little more complex. Where CMS loads programs depends on many factors:

- The mode of the virtual machine (ESA, XA, or XC).

- Whether you specify the ORIGIN option.
- The setting of the SET LOADAREA command. The default value is LOADAREA=RESPECT.
- The residency mode of the program.

Table 2 on page 50 summarizes how the various options determine where a program is loaded.

Table 2. Where CMS Loads Programs

LOAD Command	**SET LOADAREA Setting**	**Result**
LOAD pgma ...	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 largest contiguous free storage area under 16 MB.
LOAD pgma (RMODE ANY	20000	CMS loads pgma at largest contiguous free storage area above 16 MB (if available).
LOAD pgma (AMODE 24, 31 or ANY	20000	Load begins at the area determined from the default RMODE setting.
LOAD pgma (ORIGIN TRANS	20000	CMS loads pgma at the start of the transient area.
LOAD pgma (ORIGIN hexloc	20000	CMS loads pgma at hexloc.
LOAD pgma ...	RESPECT	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 16 MB. If the first RMODE setting in the TEXT file is RMODE ANY, CMS loads the program in the largest area above 16 MB. (If CMS encounters a subsequent setting of RMODE 24 in the TEXT file, it stops loading the program above 16 MB, issues a message, and starts loading the program below 16 MB.)
LOAD pgma (RMODE 24	RESPECT	CMS loads pgma at largest contiguous free storage area under 16 MB.
LOAD pgma (RMODE ANY	RESPECT	CMS loads pgma at largest contiguous free storage area above 16 MB (if available).
LOAD pgma (AMODE 24, 31 or ANY	RESPECT	Load begins at the area determined from the default RMODE setting.
LOAD pgma (ORIGIN TRANS	RESPECT	CMS loads pgma at the start of the transient area.
LOAD pgma (ORIGIN hexloc	RESPECT	CMS loads pgma at hexloc.

Note:

1. The combination AMODE 24/RMODE ANY is incorrect. Specifying AMODE 24 and an ORIGIN address greater than 16 MB is also incorrect.

2. Using the INCLUDE command:
   a. If, after you load a program above 16 MB, CMS encounters an RMODE 24 in a TEXT file you INCLUDE, CMS restarts the load process below 16 MB. Note that CMS restarts the load in the existing environment—if you change you 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 16 MB line as the program already loaded; otherwise, the INCLUDE command fails. For example, if you LOAD a program named PIE above 16 MB and you attempt to INCLUDE a program named ASLICE at an ORIGIN below 16 MB, 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 using the LOAD command, and (b) issue the START command, unpredictable results may occur.
3. The CMS loader uses the following logic when resolving names entered on the LOAD command. First search all disks for matching text file names in order from left to right. If the name is found it is loaded into storage. If it is not found, it is marked as unresolved. If text deck is loaded as a result of the name being found on a disk and it contains any VCONs, they will not be resolved until the entire command line is parsed. After the command line is parsed, unresolved symbols will be searched in the order that they where created. For more information on unresolved names see “Resolving External References by Identifying Libraries” on page 52.

How Long Does Your Program Stay in Storage?

Program life refers to how long a program remains in storage. In CMS, program life is determined by form of your program (TEXT file or module file) and by the method used to load the program.

Knowing how and when the various program forms are deleted from storage can help you select how you want to package your program and the 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 using the LOAD and INCLUDE commands remain in storage until you issue another LOAD or LOADMOD command or until CMS abend recovery occurs.
   a. You can use the PRES option of the LOAD and LOADMOD commands to prevent deletion of programs previously loaded using 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. (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 nonrelocatable 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 enter the following sequence of commands:

      ```
      load oldprog (pres
      load newprog
      ```

      Because OLDPROG has no ORIGIN specified, CMS loads it at X'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), CMS loads TEXT files at the largest contiguous area of storage available unless an ORIGIN is specified. Therefore, if LOADAREA=RESPECT you enter the following sequence of commands:

      ```
      load oldprog (pres
      load newprog
      ```

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 using the LOADMOD command remain in storage until you issue a subsequent LOAD or LOADMOD command, or until CMS abend processing occurs.

   a. You can use the PRES option of the LOAD and LOADMOD commands to prevent deletion of programs previously loaded by 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. (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 nonrelocatable program. See "1.b" on page 51 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 (end-of-command) 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 re-IPL CMS.

5. CMSCALL—if you use the CMSCALL macro to invoke a module, the module remains in storage until it completes (end-of-command) or CMS abend recovery.

6. OS/MVS LOAD macro—programs invoked using the OS/MVS LOAD macro remain in storage until they complete (end-of-command), until they are deleted by the OS/MVS DELETE macro, until they are deleted to provide storage for subsequent programs you load, or until CMS abend recovery.

   a. If you use the OS/MVS LOAD macro to load a TEXT file that (a) has already been loaded by 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 that uses the OS/MVS LOAD macro to load may be deleted if it resides in storage required by a nonrelocatable program you subsequently load.

   c. Programs that use the OS/MVS LOAD macro to load are not automatically deleted if the CMS LOAD or LOADMOD commands are subsequently issued before end-of-command.

   d. If a program is brought into storage by the LOAD macro and a LINK macro is issued 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 at the 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.

Resolving External References by Identifying Libraries

When you issue the LOAD or INCLUDE commands to load files into storage, the loader checks for unresolved references.

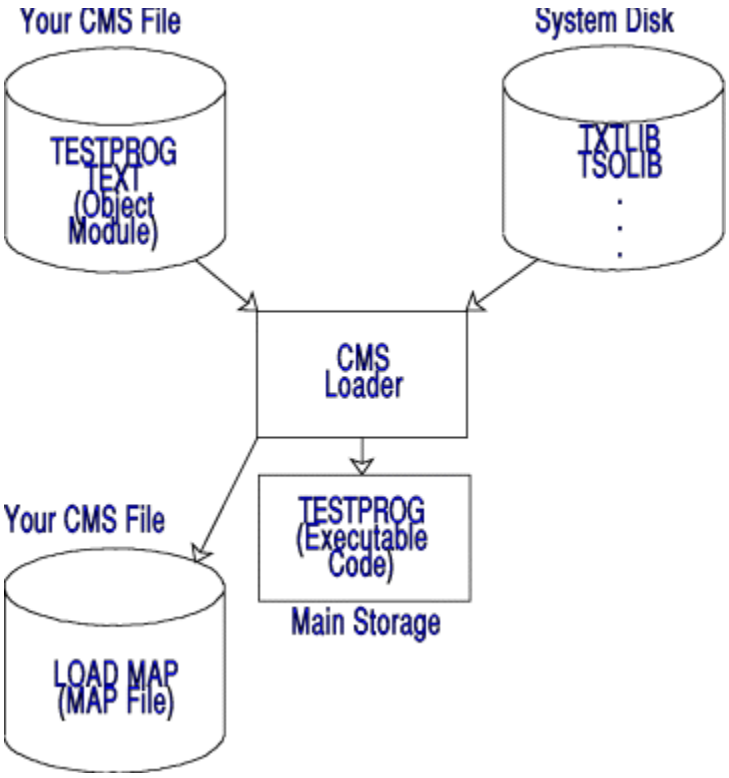


Figure 8. CMS Loader

If there are any (as a result of a CALL to a subprogram, for example), the loader searches your minidisks and directories for TEXT files with file names that match the external entry name.

Note: TEXT is the default file type searched for. You can specify a file type other than TEXT with the FILETYPE option on the LOAD and INCLUDE commands. However, for the purposes of this discussion, we will assume that you are using only TEXT files.

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, you receive a message identifying the undefined routines.

To resolve these unresolved references, issue the GLOBAL command to identify the TXTLIBs containing these routines. Then issue the INCLUDE command to load additional TEXT files or TXTLIB members into storage.

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 run 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.

A duplicate identifier message could occur if you have a library that contains a member with a name different from the CSECT name and both the member and the CSECT are forced to be loaded by unresolved references.

Example: Suppose your application called REPORTA calls routines PRINT and ANALYZE, and you enter the following LOAD command to load your application called REPORTA into storage:

```
LOAD REPORTA
```

If the loader cannot locate routines PRINT and ANALYZE, you may receive the message:

```
The following names are undefined:
 PRINT ANALYZE
```

Now, suppose NEWLIB TXTLIB contains the PRINT and ANALYZE routines. To resolve the two undefined references, enter the following GLOBAL, INCLUDE, and START commands to run your application:

```
global txtlib newlib
include print analyze
start
```

Also, you may have to identify specific TXTLIBs for the programming language you are using. For example, before running your COBOL application, you may have to enter the following GLOBAL command:

```
GLOBAL TXTLIB VSC2LTXT
```

LOAD and INCLUDE Options

There are many options of the LOAD and INCLUDE commands you may consider using. Some of these options are:

AMODE
specifies whether your application uses 24-bit addressing or 31-bit addressing. The AMODE option on the LOAD command overrides any AMODE value set on the compiler command (if an AMODE setting was used on the compiler command). AMODE is not an option on the INCLUDE command.

RMODE
specifies whether your application can reside above or below the 16 MB line. The RMODE option on the LOAD command overrides any RMODE value set on the compiler command (if an RMODE setting was used on the compiler command). RMODE is not an option on the INCLUDE command.

RESET
changes the entry point where control is passed when execution begins.

AUTO, LIBE, and DUP
controls how CMS resolves references and handles duplicate CSECT names.

CLEAR
clears storage to binary zeros before loading files. For the CLEAR option to be meaningful the ORIGIN TRANS option, which loads the program into the CMS nucleus transient area, must be used.

HIST, NCHIST
saves history information from the TEXT files. The HIST option saves all history information. The NCHIST options saves only noncommented 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.

See the *z/VM: CMS Commands and Utilities Reference* for more information on the LOAD and INCLUDE command options.

Loader Control Statements

In addition to the options provided by the LOAD and INCLUDE commands, you can also use loader control statements. You can insert these statements in TEXT files using the editor. These statements allow you to:

- Set the location counter to control the load address of the next TEXT file
- Modify instructions and constants in a TEXT file (patch a program)
- Change the entry point
- Nullify an external reference.

See the *z/VM: CMS Commands and Utilities Reference* for a description of these statements (as well as the standard loader statements produced by the compiler).

Determining Program Entry Points

When you load a single TEXT file or a TXTLIB member into storage to run, the default entry point is the first CSECT name in the object module 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.

Running Your Application

You can use one of the following methods to create an executable program and to run it:

- Use the LOAD and INCLUDE commands to produce an executable program in virtual storage. Use the START command to run this program.
- Use the LOAD, INCLUDE, and GENMOD commands, or use the BIND command, to build an executable program stored as a MODULE file on a CMS disk or directory. Run the program by issuing the name of the MODULE file created by the GENMOD or BIND command.
- Use the LKED or BIND command to create an executable program stored as a load module in a member of a CMS LOADLIB. Use the OSRUN command to run this program.
- Use the LOAD, INCLUDE, and GENMOD commands, or use the BIND command, to build an executable program stored as a module file on a CMS disk or directory. Use the OPENVM PUTBFS command to copy the file to BFS. Use the OPENVM RUN command to run this program.
- Use the c89 utility, the cxx utility, or the BIND command to create an executable file in BFS. Use the OPENVM RUN command to run this program.

Using the START Command

After you compile your application, access any libraries using the GLOBAL command, and create a temporary copy of your executable program in virtual storage using the LOAD and INCLUDE commands, issue the START command to run your application.

Example: Suppose you have an object file called WELCOME TEXT. You can run this program using the following sequence of commands:

```
load welcome
start
```

or the following command:

```
load welcome
(start
```

Passing Parameters on the START Command

If the program you are going to run expects a parameter list, you can specify the arguments on the START command line. For example, to pass the argument 007 to your application, enter:

```
start * 007
```

* specifies the default entry point that is usually the beginning of your application. See the *z/VM: CMS Commands and Utilities Reference* for details on the START command.

Using the GENMOD Command

After you compile your application, access any libraries needed using the GLOBAL command, and load your application into virtual storage using the LOAD and INCLUDE commands, issue the GENMOD command to create a MODULE file stored on your disk or directory. Then, issue the file name of the MODULE file to run your application.

Example: Suppose you have an object file called WELCOME TEXT. To create a MODULE file, you must load the required files into storage and then enter the GENMOD command:

```
load welcome
genmod welcome
```

Now, CMS generated an executable nonrelocatable program called WELCOME MODULE. See "Creating Nonrelocatable and Relocatable Modules" on page 56 for information about creating relocatable and nonrelocatable MODULE files.

To run WELCOME MODULE, enter:

```
welcome
```

If your program expects arguments passed to it, you can enter them following the module name. For example, if you want to pass WELCOME MODULE the argument 007, enter:

```
welcome 007
```

Creating Nonrelocatable and Relocatable Modules

A relocatable module is one that CMS does not need to load at a specific storage location. CMS loads relocatable modules at the highest available storage range large enough to contain it. Defining your modules as relocatable helps eliminate the possibility that the storage your module requires is being used by another program. By contrast, CMS loads nonrelocatable modules according to the location of the TEXT file when the module was created.

For CMS to create a relocatable module, you must specify the RLDSAVE option on the LOAD command. The RLDSAVE option instructs the CMS loader to save the relocation information from the TEXT file. By default, CMS does **not** save this relocation information when you use the LOAD command.

Example — Creating a Relocatable Module

To create a relocatable module named OZ from TEXT files named DORTHY, TINMAN, LION, and TOTOTOO, enter the following commands:

```
load dorthy tinman lion tototoo (rldsave
genmod oz
```

When you run OZ MODULE, CMS loads it at the highest available storage range large enough to contain it.

Example — Creating a Nonrelocatable Module

To create a nonrelocatable module named NOTOZ from a TEXT file named WITCH TEXT, enter the following commands:

```
load witch (norldsav
genmod notoz
```

Or, because NORLDSAV is the default value, you could enter:

```
load witch
genmod notoz
```

When you run 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 enter the GENMOD command. For example:

```
load myprog (origin trans
genmod
```

The two restrictions placed on command modules running in the transient area are:

- They may have a maximum size of 8192 bytes (the size of the transient area).
- They must be serially reusable.

The *z/VM: CMS Commands and Utilities Reference* identifies the CMS commands that run in the transient area.

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 file. 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 XC Mode

You can use the XC option of the GENMOD command to specify that a module will run only in an XC virtual machine. For example, to specify that ODDJOB will run only in an XC virtual machine, enter:

```
load oddjob
genmod (XC
```

Note: CP does not support System/370 (370 mode) virtual machines. The 370 option of the GENMOD command is also not supported. However, the CMS SET GEN370 OFF command allows modules generated with the GENMOD 370 option to run in an ESA, XA, or XC virtual machine. See the *z/VM: CMS Commands and Utilities Reference* for more information on the SET GEN370 command.

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 all history information. The NCHIST option saves only noncommented 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 history information that was in PROGONE TEXT and PROGTWO TEXT.

Loading MODULE Files

To load a MODULE file, you can:

- Issue the LOADMOD command from your terminal, from an EXEC, or by the CMSCALL macro from an assembler 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. The LOADMOD command should be used only with module files created by the GENMOD command with the MAP option.
- Enter the name of the module from your terminal.
- Issue the NUCXLOAD command from your terminal, from an EXEC, or by the CMSCALL macro from an assembler program.
- Issue the CMSCALL macro from an assembler program.
- Issue the OS/MVS LOAD macro from an assembler program.

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 27, “Using Saved Segments,” on page 419. For information about defining and building saved segments, see *z/VM: CP Planning and Administration*.

Note: Building a saved segment requires the CP authority to perform the DEFSEG and SAVESEG operations.

Using the BIND Command

After you compile your application and access any libraries needed using the GLOBAL command, use the BIND command to create a module file stored on your disk or directory. Then enter the file name of the module file to run your application. The BIND command may also be used to create an executable file in BFS or a member of a CMS LOADLIB. For more information about using the BIND command, refer to *z/VM: Program Management Binder for CMS*.

Using the LKED and OSRUN Commands

After you compile your application, you can create (also referred to as link-edit) an executable application using the LKED command. The LKED command uses the MVS/XA linkage editor to create this executable program from a CMS TEXT file, a TXTLIB member, or another LOADLIB member and stores it as a load module as a member of a CMS LOADLIB.

The primary LKED input is a data set known to the linkage editor as SYSLIN, which is identified by the *fname* operand of the LKED command. The file type of the input file named 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 data set may be:

1. An object module (TEXT file), such as, assembler or compiler output
2. Linkage editor control statements
3. A combination of an object module 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 data set 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.

Once you have identified all your input files and created your executable program, issue the GLOBAL and OSRUN commands. The GLOBAL command identifies all the load libraries that need to be searched when running your program. The OSRUN command runs your load module.

Example 1: Suppose you have a FORTRAN object file called TESTFILE TEXT. To create the CMS LOADLIB, TESTFILE LOADLIB, enter:

```
FILEDEF SYSLIB DISK VSF2FORT TXTLIB *
LKED TESTFILE
```

The FILEDEF SYSLIB command resolves any FORTRAN library routines references when creating the LOADLIB file. 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 details on CMS OS data management, see the *z/VM: CMS Application Development Guide for Assembler*.

The linkage editor produces two permanent files on your A-disk. The file name of both files is the name specified on the LKED command. One file contains the load module(s) created by the linkage editor. It is given the file type LOADLIB. The other file is the printed output from the linkage editor. It is given the file type LKEDIT. If you specify the PRINT or NOPRINT option on the LKED command, the LKEDIT file is not created on disk. The LKEDIT file is sent directly to the printer.

Before running TESTFILE, use the GLOBAL command to identify the LOADLIB to be searched and any other LOADLIB necessary because of the programming language you are using. For example:

```
GLOBAL LOADLIB TESTFILE VSF2LOAD
```

Then, the following OSRUN command performs the search and loads, relocates, and runs the TESTFILE member of TESTFILE LOADLIB:

```
OSRUN TESTFILE
```

The OSRUN command searches only the libraries specified in the GLOBAL LOADLIB command, unless you have a system library named $SYSLIB LOADLIB. Then, OSRUN searches $SYSLIB LOADLIB for the member name.

Example 2: Linking a Program that Requires More than One Library: Suppose you have a VS FORTRAN object module called TEST TEXT that explicitly invokes a routine called SUB0000 from a user library called USERLIB TXTLIB and implicitly invokes routines from the VS FORTRAN library called VSF2FORT TXTLIB. 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 TXTDEF
LIBRARY LIBDEF(SUB0000)
```

Next, enter the following commands:

```
FILEDEF TXTDEF DISK TEST TEXT A
FILEDEF SYSLIB DISK VSF2FORT TXTLIB *
FILEDEF LIBDEF DISK USERLIB TXTLIB *
LKED NPUT (LIBE TESTLIB NAME TEST0000
```

To run TEST0000, enter the following:

```
GLOBAL LOADLIB TESTLIB VSF2LOAD
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 VSF2FORT
FILEDEF SYSLIB CLEAR
FILEDEF SYSLIB DISK USERLIB TXTLIB * (CONCAT
LKED TEST (LIBE TESTLIB NAME TEST0000
```

To run TEST0000, enter the following:

```
GLOBAL LOADLIB TESTLIB VSF2LOAD
OSRUN TEST0000
```

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

LKED Options

The LKED command has many options available for you to use. CMS does not use all of the options. The CMS-related options are:

TERM, NOTERM
causes the linkage editor to display diagnostic messages or to suppress such messages at the terminal. TERM is the default option.

PRINT, DISK, NOPRINT
directs the linkage editor printed output to specific medium. PRINT spools the linkage editor printed output to the printer. DISK stores the linkage editor output in a CMS disk files with a file type of LKEDIT. DISK is the default option. NOPRINT suppresses all printed output.

AMODE
specifies whether your application can run in 24-bit addressing mode or 31-bit addressing mode. This option overrides the AMODE setting on the LOAD command.

RMODE
specifies whether your application can reside above or below the 16 MB line.

NAME *membername*
identifies the member name to be used for the load module created.

LIBE *loadlibname*
identifies the file name of a LOADLIB file where the load module is placed.

You can use the following options with the linkage editor to specify characteristics of the load module: LET, NE, OL, RENT, REUS, REFR, and OVLY. XREF, MAP, LIST, NCAL, XCAL, SIZE, and ALIGN2 are also options of the LKED command. See the *z/VM: CMS Commands and Utilities Reference* for a description of these options.

Using the OPENVM RUN Command

You can use the OPENVM RUN command to invoke applications that reside in the BFS or record file system or exist as nucleus extensions. Files invoked by OPENVM RUN must have been generated with the GENMOD command, the BIND command, the c89 utility, or the cxx utility. Because of the multitasking nature of OpenExtensions applications, these modules must be generated to be relocatable in order to avoid overlay problems. When using the OPENVM PUTBFS command to copy a module file from a minidisk or SFS directory to BFS, the MODULE option must be used (or defaulted to) to make sure that the BFS file is in executable format.

The OPENVM RUN command accepts a path name as input. The path name can be the name of a module file in the BFS, an external link to a module file in the record file system, or the CMS file ID of a module file in the record file system or loaded as a nucleus extension.

OPENVM RUN first tries to open the path name as a file in the BFS. If the open is successful, this is the file that is loaded and invoked. If the file is an external link, OPENVM RUN obtains information about the file that the external link represents, including the CMS file ID. This file is invoked if it is an FST_EXEC type of external link and one of the following is true:

- The file resides on an accessed minidisk or SFS directory.
- The file type of the file is MODULE or unspecified, the file mode of the file is * or unspecified, and the file is loaded as a nucleus extension.

If neither a suitable BFS file nor an external link is found, OPENVM RUN attempts to interpret the path name as a CMS file ID. If the path name can successfully be parsed into a CMS file ID, OPENVM RUN determines if the specified file exists on the accessed file modes, or, under the conditions described above, whether it is loaded as a nucleus extension. If the file is found, this is the file that is invoked. Otherwise, the OPENVM RUN command fails.

Note that when using the OPENVM RUN command, files in the record file system need not have a file type of MODULE. Renaming a MODULE file to give it a different file type is one way to prevent an OpenExtensions application in the record file system from being invoked by name from the CMS command line. It should also be noted that the OPENVM RUN command is case sensitive, so you have to enter the file ID of a file in upper case if its name is in upper case. In addition, path names that contain blanks must be delimited by quotation marks. If you want to specify a file name and file type on the OPENVM RUN command, you must enclose the entire CMS file ID in single or double quotation marks.

For example, if you want to invoke the OpenExtensions application MY OEAPP that resides on your A disk, enter:

```
openvm run 'MY OEAPP'
```

Passing Parameters on the OPENVM RUN Command

The OPENVM RUN command provides the ability to pass parameters to the application. Each parameter specified on the command has a NULL character (X'00') appended to it and is passed to the application as a separate argument. Parameters that contains blanks or special characters, such as quotation marks, must be enclosed in quotation marks. (For more information, see the description of the OPENVM RUN command in the *z/VM: OpenExtensions Commands Reference*.) The first parameter that OPENVM RUN passes to the application is always the file name. Any user-specified parameters come after the file name.

For example, if you want to invoke the application called /u/FamilyTree and pass it the parameters "DAUGHTER Megan" and "SON Brian", enter:

```
openvm run /u/FamilyTree 'DAUGHTER Megan' "SON Brian"
```

This would invoke the file /u/FamilyTree with three parameters:

- FamilyTree
- DAUGHTER Megan
- SON Brian

The parameter list that OPENVM RUN passes to the application is pointed to by register 1. It is not a standard or tokenized parameter list, but is the type that the exec (BPX1EXC) callable service passes. (For more information about this parameter list, see the description of the exec (BPX1EXC) service in the *z/VM: OpenExtensions Callable Services Reference*.) An application can check the USECTYP flag in the user save area that is pointed to by register 13 to determine what type of parameter list it is getting. The user save area can be mapped using the USERSAVE macro. If the flag contains a X'10', the parameter list is the type passed by the exec (BPX1EXC) callable service. C or C++ applications need not worry about this, because the run-time code determines what kind of parameter list is being passed, and the application code is entered as follows:

```
int main (int argc, char *argv[]);
```

where *argc* is the argument count and *argv[]* is an array of character pointers to the arguments themselves.

Things to Be Aware of When Using OPENVM RUN

It is not possible to specify run-time options on the OPENVM RUN command. If an application needs run-time options, another language-specific method of passing them must be used.

The OPENVM RUN command initializes the POSIX environment variables from the GLOBALV variables in the CENV group. In addition, it defines the LOGNAME, HOME, PATH, and SHELL environment variables in a POSIX-compliant manner. For more information on setting these variables, see the description of the OPENVM RUN command in the *z/VM: OpenExtensions Commands Reference*.

Because many OpenExtensions applications expect STDIN, STDOUT, and STDERR to be opened as file descriptors 0, 1, and 2, respectively, OPENVM RUN opens the terminal if any of these file descriptors is not already open.

When OPENVM RUN loads a module into storage prior to execution, it copies any map information that was saved at GENMOD time into the loader tables. However, it is unwise to rely on the loader table information, because the loader tables are shared among all of the processes in the virtual machine, and the information can be overwritten whenever another module file is loaded.

Displaying Information about Programs In Storage

You can use the PROGMAP command to display information about programs currently loaded in storage.

PROGMAP Command

The PROGMAP command obtains the name, entry point, origin, addressing mode, and relocation attributes of programs that were loaded using the LOAD, INCLUDE, LOADMOD, or NUCXLOAD commands or the OS LOAD macro.

If you issue PROGMAP from within a program, use the STACK and LIFO|FIFO 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
PROG3    02000E97    02000E97    00000338    Amode 31  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       02000E97    02000E97    00000338    Amode 31 Reloc
Name      Entry    Userword Origin   Bytes   Amode (Attributes)
NUCX1     0035A000 00000000 00000000 00000000   31  SYSTEM SERVICE
NUCX2     0035E934 00361828 00000000 00000000  Any  SYSTEM
NUCX3     004DB000 00000000 004DB000 00001FF8   24  SYSTEM SERVICE        IMMCMD
 PERM
```

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 7. Debugging and Testing Your Program

This chapter describes commands, tools, and facilities you can use to help debug and test your applications. These topics include:

- CP and CMS commands that provide access to general registers, main storage, and control words, as well as trace options and dump control
- Debugging tools available for some of the programming languages
- Dialog and testing service of ISPF
- Using SQL for database prototyping and testing
- Testing your complete application package.

Commands Used for Debugging

You can use these the CP and CMS commands to debug your applications:

CP Commands for Debugging

Table 3. CP Commands for Debugging

CP Command	Description
CPEREPXA	Accesses the Environmental Record Editing and Printing (EREP) program. This program reads EREP records from a disk or a tape and produces printed output reports.
DISPLAY	Displays at your terminal the contents of your virtual machine registers, virtual machine storage, old and new PSWs, storage keys, or subchannel information blocks.
DUMP	Dumps to your virtual printer the contents of your virtual machine registers, virtual machine storage, old and new PSWs, storage keys, and subchannel information blocks.
MONITOR	Controls the selection, collection, and reporting of data from the system.
QUERY CPTRACE	Displays the current setting of the tracing of real system events.
QUERY RECORDING	Determines the status of CP data collection for accounting, symptom, and EREP records, and determines if record retrieval is in progress.
QUERY TRSAVE	Displays the destination of traces defined by TRSOURCE or displays the status of traces controlled by the SET CPTRACE command.
QUERY TRSOURCE	Displays the current status of the traces defined using the TRSOURCE command.
RECORDING	Alters the processing parameters for CP recording facilities.
RETRIEVE	Collects accounting, EREP, and symptom records and places the records into files for later processing and analysis.
SET CPTRACE	Activates or deactivates the tracing of real machine events.
SET MODE	Sets the error recording mode for system recovery machine checks.
SET RECORD	Sets the recording mode for a device.
STORE	Alters virtual storage locations, registers, PSWs, CAWs, and CSWs.

Table 3. CP Commands for Debugging (continued)	
CP Command	**Description**
TRACE	Monitors events that occur in your virtual machine. You can monitor: execution of instructions, changes to storage, changes to registers, and I/O activity.
TRSAVE	Specifies where CP trace table data or data from traces defined by the TRSOURCE command will be saved.
TRSOURCE	Defines a trace (an I/O, data, or guest trace) and controls (enables, disables, drops, or displays) the individual trace.
VMDUMP	Dumps all or selected pages from second level storage. This information can be used by the dump viewing facility. VMDUMP also saves the following information: virtual program status word, general registers, control registers, storage protection keys, and timer values.

For more information on these CP commands, see the *z/VM: CP Commands and Utilities Reference*.

CMS Commands for Debugging

You can use these CMS commands to debug your applications:

Table 4. CMS Commands for Debugging Applications	
CMS Command	**Description**
DEBUG	Displays status information following ABEND processing.
MODMAP	Displays the load map associated with the specified MODULE file.
PROGMAP	Displays information about programs currently loaded in storage.
STDEBUG	Traces the obtain and release requests made by an application.
STORMAP	Displays storage information about your virtual machine. This information includes the amount of allocated and unallocated free storage within your virtual machine and the size of the largest contiguous block of storage both above and below the 16 MB line.
SUBPMAP	Displays storage allocation information for subpools in your virtual machine.
SVCTRACE	Provides you with a record of all supervisor calls in your virtual machine. The information, which is routed to your printer, includes: • Call and return address information. • GPR and floating-point register contents before, during, and after the call. If you have more than one printer available, you may want to route the trace information to a separate printer from your program output. Depending on the type of problem, sometimes it is more informative to intermix the two outputs. For details on the SVCTRACE command, see the *z/VM: CMS Commands and Utilities Reference*.

Interactive Debug Tools for Specific Languages

The Interactive Debug tools contain a command and a set of subcommands that help you in diagnosing and solving problems in your programs. These tools let you:

- Stop and start the program as it runs
- Examine and change values of variables

- Trace program transfers
- Track frequency of execution of statements
- Locate errors and correct them
- Test the code and improve its efficiency.

Debugging Your COBOL Application

You can use COBTEST, the VS COBOL II debug tool, to debug any VS COBOL II program. You can debug your applications in batch mode or interactive mode. When debugging your application interactively, you can use line mode or full-screen mode.

If you use COBTEST in full-screen mode, you must use the Interactive System Productivity Facility (ISPF), Version 2. You must also specify the TEST compiler option to use COBTEST.

For details on debugging VS COBOL II applications, see the *VS COBOL II Application Programming: Debugging Guide*.

Debugging Your FORTRAN Application

You can use the VS FORTRAN Version 2 Interactive Debug tool to debug VS FORTRAN Version 1 and VS FORTRAN Version 2 programs. You can debug your applications in batch mode or interactive mode. When debugging your application interactively, you can use line-mode or full-screen mode.

Invoking VS FORTRAN Interactive Debug in full-screen mode requires Interactive System Productivity Facility (ISPF) Version 2, with or without the ISPF/Program Development Facility (ISPF/PDF).

For details on debugging your VS FORTRAN applications, see the *VS FORTRAN Version 2 Interactive Debug Guide and Reference*.

Debugging Your Pascal Application

VS Pascal provides you with an interactive debugging tool that allows you to debug your VS Pascal applications without having to write debugging statements directly into your source program. You can use this tool in interactive mode and in batch mode.

To invoke this interactive debugging tool, follow these steps:

1. Use the DEBUG option on the compiler command.
2. Before you link-edit your application, use the GLOBAL command to identify the debugging library and run-time library needed for the interactive debugging tool. You should identify the debugging library before the run-time library. When you invoke the PASCMOD exec with the DEBUG option to build your load module the debugging library, PASDEBUG TXTLIB, is automatically identified.
3. Use the DEBUG run-time option when you execute the load module.

Once you are in the debugging environment, you can issue the debug commands provided by the VS Pascal language.

For details on the interactive debugging tool, see the *VS Pascal Application Programming Guide*.

Dialog Testing Using ISPF

The dialog test option provides you with aids for testing ISPF dialog parts (functions, panels, variables, messages, tables, skeletons) and complete ISPF applications. For example, you can:

- Invoke selection panels, command procedures, programs and shared segments
- Display panels
- Add new variables and modify variable values
- Display a table's structure and status
- Display, add, modify, and delete table rows

- Browse the ISPF log, provided that ISPF/PDF is installed
- Execute dialog services
- Add, modify, and delete function and variable trace definitions
- Add, modify, and delete breakpoint definitions.

When you enter dialog test from the ISPF/PDF primary option panel, you enter a new user application with an application ID of ISR. When you enter dialog test from the ISPF primary option panel, you enter a new user application with an application ID of ISP. All the options operate in this context.

Dialog test is itself a dialog and, therefore, uses the dialog variables. Because it is important to allow your dialog to operate without interference (as though in a production environment), dialog test accesses and updates variables independently of your dialog variables.

If your dialog encounters a severe error when it invokes a dialog service, that error is handled as requested by a dialog. The current CONTROL service ERRORS setting (CANCEL, or RETURN; the default is CANCEL) determines what is done. If CANCEL is in effect, when the error message panel is displayed you may choose whether to continue dialog testing.

The following is a description of the test options:

The **functions option** lets you test a dialog function (panel, command procedure, or program). You do not have to write supporting code or panels. The name of the dialog function and the parameters that may be passed are the same as those that you can specify (from a dialog function) when you invoke the SELECT service. When you press Enter, a SELECT is done. When you select this option, a panel is displayed that lets you identify the dialog function that you want to test.

During panel development, the **panels option** lets you test newly created or modified panels and messages. You do not need to write supporting code to display them. Any variables referenced and set during panel processing are handled according to standard ISPF protocol.

The **variables option** lets you:

- Display all ISPF variables defined in the dialog application you are testing
- Change the value of a variable
- Define new variables
- Delete variable names and blank lines.

When you select this option, a scrollable display indicates all the current variables for the dialog being tested. The rows of the display are ordered by the pool containing the variables, then by function pool type within the function pool, then alphabetically by variable name within each pool. The function variable pool is listed first, followed by the shared variable pool, and then the profile variable pool. Insertions are left where they are entered on the display.

Modifications to the display are processed when you press Enter. Updating of the variable pools occurs when you enter the END command.

The **tables option** lets you:

- Display the contents of an existing row in an open table.
- Remove an existing row from an open table.
- Change the contents of an existing row of an open table.
- Add a new row after a selected row of an open table.
- Display the structure of a table.
- Display a data information panel reflecting all operations using a specified table.

The **log option** lets you display and browse data recorded in the ISPF log. The ISPF/PDF product is required for this function. You can use all the browse commands, except BROWSE, while looking at the ISPF log. The ISPF log contains the following types of trace output:

- Trace header entries
- Function trace entries

- Variable trace entries.

The **dialog services option** lets you execute a dialog service by entering the service command invocation with or without the ISPEXEC characters. You can call any dialog service that is valid in the command environment except CONTROL at a breakpoint or before invoking a function.

The **traces option** lets you define, change, and delete trace specifications. You can trace executed dialog services, except for the VPUT or VGET service issued to a panel, and referenced dialog variables during dialog execution. Trace data is placed in the transaction log. From here you can browse it (using the LOG option), or print it when you exit from ISPF.

Because tracing may degrade dialog performance and create large amounts of output, care should be taken in setting the scope of trace definitions.

When you select this option, you are shown a selection panel on which you can indicate the type of trace (function or variable) you wish to define.

Use the **function traces option** to establish criteria for recording the names of dialog service calls, the service parameters, and return code in the ISPF log. Service calls made by the dialog or during test processing are recorded. Whenever a new application or function has data recorded, a header is placed in the trace. When you select the function traces option, a scrollable panel displays all currently defined function traces. You may add, delete, and modify function trace definitions using this panel before invoking a function or at a breakpoint.

The **variable traces option** establishes criteria for recording variable usage. The usage of a variable is recorded:

- If an ISPF service is directly asked to operate on the variable (for example, VGET, VPUT, VCOPY).
- If an ISPF service is indirectly asked to operate on the variable (for example, DISPLAY).

  Variables changed under the variables option are also recorded if the trace specifications are met.

When you select the variable traces option, a scrollable display lists all currently defined variable traces. You may add, delete, or modify variable trace definitions by using this panel before invoking a function.

A breakpoint is a location at which the execution of a dialog is suspended so that dialog test facilities may be used. The **breakpoint option** lets you indicate where such temporary suspensions should occur. At a breakpoint, you are given control. You may now examine and manipulate dialog data (for example, tables and variables) using various test options. You can also specify new test options, such as traces and other breakpoints.

Breakpoints are located immediately before a dialog service receives control or after it relinquishes control. Breakpoint definitions cause special handling within the ISPLINK, ISPLNK, and ISPEXEC interfaces to dialog services. No user dialog is modified.

When you select the breakpoint option, a scrollable display shows all currently defined breakpoints for this session. You may add, delete, or modify breakpoint definitions using this panel before invoking a function or a breakpoint. All breakpoints exist until you delete them or you end or cancel your dialog test session. If you invoke a dialog function or a selection panel and encounter a breakpoint, the dialog test breakpoint primary option menu is displayed.

Like the dialog test primary option menu, the breakpoint primary option menu lets you use the RETURN command from any one of the selected test options to process a redisplay of the breakpoint primary option menu. You must use:

- The GO option to terminate processing at this breakpoint and continue executing the dialog being tested.
- The CANCEL option to cancel the dialog test option.

The breakpoint primary option menu contains all options of the dialog test primary options menu except Exit. It therefore presents all but one of the dialog test functions to you.

When a user dialog encounters a breakpoint, the current dialog environment is saved. When you select the GO option, the environment is restored, except for the following:

- If you change variable, table, and file tailoring data at a breakpoint, these actions are performed as an extension of the suspended dialog. It is as if the dialog takes all the actions itself during execution.
- If you modify the service return code (on the breakpoint primary option menu), the new return code is passed back to the dialog. It is as if the service sets the new code itself.
- If you execute the PANELID command at the breakpoint, the last setting for displaying panel identifiers is retained.
- If any CONTROL service settings for DISPLAY LINE or DISPLAY SM are in effect before the breakpoint, such settings are lost.

The manipulation of one dialog part may cause a change to another dialog part.

For further information on these functions and all dialog test functions, see the *ISPF Dialog Management Guide and Reference*.

Database Testing Using SQL

You can use SQL as a tool for prototyping data designs and implementations during the application development process. For example, the ability to CREATE, ALTER, and DROP tables dynamically from an online, interactive environment lets you experiment with different designs.

SQL facilities support these data prototyping functions:

- Online definition of model designs
- Generation/loading of test data
- Design documentation and analysis
- Sharing host variables or SQL commands.

Online Definition of Model Designs: You can use ISQL to enter table, view, and index definitions for validating and testing data design. The interactive definition through ISQL offers you direct feedback on definitional errors. This feedback addresses both syntax and data mapping errors.

If you issue SQL definitional commands using ISQL, then it can save them as stored queries for later recall, modification, or rerun. You can also save statements in CMS files used as input (SYSIN) to the DBS utility.

Generation/Loading of Test Data: You can load tables created for design purposes with test data using these SQL facilities:

- Item by item, using the ISQL INPUT command.
- From existing SQL tables within the database, using the SQL INSERT command.
- From existing SQL tables in another database, using the DBS UNLOAD and RELOAD commands.

Design Documentation and Analysis: By using the SQL explanation tables and the EXPLAIN command, you can analyze how a given design will perform. You can issue the EXPLAIN command using ISQL, the DBS utility, or an application program. EXPLAIN lets you get information about the structure and execution performance of a SQL command.

You can see how well a SELECT command performs by using the ISQL query cost estimate. ISQL displays this at the end of every SELECT result. This estimate of the resources used during command execution is related to, but is not the same as, that obtained by EXPLAIN.

Sharing Host Variables or SQL Commands: When you are developing programs, you may want to use the SQL INCLUDE command. This is useful when many applications use the same host variables or SQL command sequence. This command causes the preprocessors to include source lines from other CMS files in your source program.

Suppose you have a lengthy SELECT command that many programs use. First, place this SELECT command in a separate CMS file, called SOURCE1, for example. Then, in your source program, put the following SQL statement where you want to include the SELECT command:

```
EXEC SQL INCLUDE SOURCE1 END-EXEC.
```

When developing a program with embedded SQL commands, you can run the SQL preprocessors with a CHECK option. This causes the preprocessor to generate diagnostics on the SQL in the program but not an access module or compiler input. You can thus use a skeleton of the final program to do a lot of initial code development and debugging.

Using ISQL

You can use ISQL facilities to test and debug SQL commands for application development. The ISQL support of routines lets you develop logical sequences of SQL commands for this purpose. You can produce different routines using parameters to simulate program variables for various paths through the application logic. This tests the functional results of an application against various inputs.

In these situations, you can use the ISQL command SET RUNMODE, which lets you stop or continue the execution of an ISQL routine when an error occurs.

This command offers these options:

- Continue to the next command even if an error occurs. (You can use this option to bypass unconnected errors or examine later ones.)
- Stop processing when you make an error, but do not perform ROLLBACK WORK (that is, leave the data in its processed state).
- Stop processing when you make an error, but perform ROLLBACK WORK (that is, erase all changes the routine made and preserve the integrity of the database).

Testing Your Complete Application Package in a Virtual Machine

The virtual machine environment is an ideal environment to test a complete application package. Within the same z/VM system, you can test your application package in a virtual machine that is completely isolated from an "active" production environment.

The virtual machine concept of z/VM also allows you to set up z/VM as a second level system. That is, z/VM can be the operating system of a real machine or as the operating system of a virtual machine. The following figure shows the relationship between a first level system and a second level system.

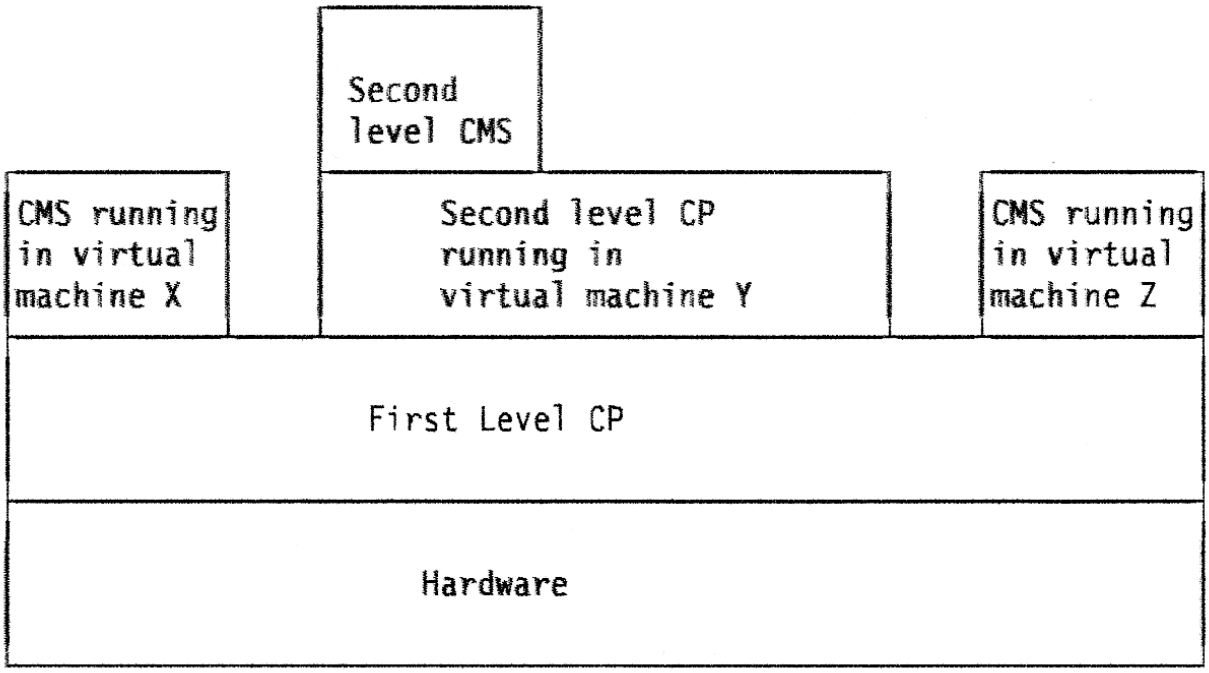


Figure 9. Relationship between First Level and Second Level Systems

As an example, suppose you build a second level system to test your application and you are a privilege class user. While testing your application, if you issue a command that crashes your second level system, you will not disturb the function or users of the first level system.

See *z/VM: Running Guest Operating Systems* for information on setting up a second level system.

Chapter 8. Updating Your Source Program

This chapter discusses the following topics:

- Making updates to a source file using an UPDATE file
- The contents of an UPDATE file
- Making multiple updates to a source file using a control file and an auxiliary control file
- Making updates to exec and macros
- Writing your own exec to invoke the UPDATE command
- An example of updating a program.

The simplest way to maintain backup copies of a program is to make a copy of the current source file under a new name. You can do this using either the COPYFILE command or the editor. While this procedure for modifying programs is suitable for many applications, it may not be adequate in a situation where several programmers are applying changes to the same source code. Also, this procedure does not provide you with a record of what has been changed. After using the editor, you do not have a record of the lines that have been deleted, added, replaced, and so on, unless you manually add comments to the code, insert special characters in the serialization column, or use some technique that records program activity.

The XEDIT command with the UPDATE option and the UPDATE command allow you to:

- Modify the source program without affecting the original source file
- Record all the changes in separate UPDATE files
- Time-stamp and identify the changes in the UPDATE files
- Apply and remove changes as needed.

This update process allows you to apply one UPDATE file to a source file, one set of UPDATE files to a source file, or more than one set of UPDATE files to a source file. The XEDIT command and the UPDATE command use four files to during the update process:

- Source file (fn ft fm)
- UPDATE file (fn UPDATE fm)
- Updated source file ($fn ft fm)
- Update log file (fn UPDLOG fm).

Making Updates to a Source File

To update a source file following these steps:

1. Use the XEDIT command with the UPDATE option. The XEDIT command makes changes to the source file by creating an UPDATE file that contains the new or changed source statements and the update control statements.
2. Use the UPDATE command. The UPDATE command incorporates the changes recorded in the UPDATE file and produces an updated source file and an update log file.

Step 1 - Using the XEDIT Command to Make Changes to a Source File

The XEDIT command with the UPDATE option creates an UPDATE file that contains the changes made to the source file. This way the original source file is not affected.

Creating an UPDATE File

When using the XEDIT command to create an UPDATE file, the editor expects the source files to have sequence numbers in the last eight columns of each record. The logical record length of the source file can be up to 255. The source file has a fixed record format.

However, if you use the XEDIT subcommand SET SERIAL to sequence your files or if the file has been sequenced using the default values of XEDIT, the sequence numbers are usually written in the last five columns of the logical record length and prefaced by the first three character of the file name.

Therefore, depending on the location of the sequence numbers, you can create an UPDATE file using the following two commands:

- The XEDIT command with the UPDATE option.
- The XEDIT command with the UPDATE option and the NOSEQ8 option.

Using the XEDIT Command With the UPDATE Option

Using the XEDIT command with the UPDATE option to create an UPDATE file, the editor expects the source files to have sequence numbers in the last eight columns of the file. To generate sequence numbers, edit your file, and before issuing the FILE or SAVE subcommand, enter the following XEDIT subcommand:

```
serial all
```

Now, to make changes to the source file and to create an UPDATE file. enter the following command:

```
xedit ready cobol a (upd
```

This command specifies that a source file called READY COBOL is to be edited, but all updates to the file are placed in a separate UPDATE file called READY UPDATE along with the appropriate control statements. Using XEDIT, you do not need to enter the control statements in the UPDATE file. For details on the control statements, see “Using a Control File” on page 78. They are generated automatically by the editor.

Using the XEDIT Command With the UPDATE Option and the NOSEQ8 Option

Using the XEDIT command with the UPDATE option and the NOSEQ8 option to create an UPDATE file, the editor expects the source file to have sequence numbers in the last five columns.

For example:

```
xedit ready cobol a (upd noseq8
```

specifies that a file called READY COBOL is to be edited and all updates to the file are placed in a separate UPDATE file called READY UPDATE along with the appropriate control statements. Using XEDIT, you do not need to enter the control statements in the UPDATE file. For details on the control statements, see “Using a Control File” on page 78. They are generated automatically by the editor.

Using an Existing UPDATE File

If an UPDATE file already exists for a given source file, you can continue updating the source file using the same UPDATE file. For example, if READY UPDATE already exists and you want to make additional changes to the source file, enter the following command:

```
xedit ready cobol a (upd
```

This command applies all updates contained in READY UPDATE to the source file READY COBOL and displays the resulting file on the screen. Now, you can make other updates created during this editing session and these updates are added to those already contained in READY UPDATE. Again, all control statements are automatically generated by XEDIT.

Step 2 - Using the UPDATE Command to Add Changes to a Source File

Use the UPDATE command to add the changes from the UPDATE file to create a new updated source file. The UPDATE command produces an update log indicating the changes that have been made.

The default values used by the UPDATE command are file types of ASSEMBLE and UPDATE for the source and update files, respectively. If you are updating a COBOL source program named READY COBOL with an UPDATE file named READY UPDATE, you would enter the command:

```
update ready cobol a ready update a
```

After an UPDATE command completes processing, the input files are not changed. Two new files are created. One of them contains the updated source file, with a file name that is the same as the original source file but preceded by a dollar sign ($). Another file, containing a record of updates is also created. It has a file name that is the same as the source file and a file type of UPDLOG. For example:

Source Files
Output Files

SAMPLE ASSEMBLE
$SAMPLE ASSEMBLE

SAMPLE UPDATE
SAMPLE UPDLOG

READY COBOL
$READY COBOL

READY UPDATE
READY UPDLOG

When using the UPDATE command, the editor expects the source files to have sequence numbers in the last eight columns of the file.

However, if you use the XEDIT subcommand SET SERIAL to sequence your files or if the file has been sequenced using the default values of XEDIT, the sequence numbers are usually written in the last five columns of the file and prefaced by the first three character of the file name.

Therefore, depending on the location of the sequence numbers, you can create an updated source file the following ways:

- serial all
- ./ S
- update (noseq8

serial all: If you want an eight-character sequence number and you are editing the file, you must use the subcommand:

```
serial all
```

before issuing a FILE or SAVE subcommand.

./ S: You can create an UPDATE file with the single record:

```
 ./ S
```

and issue the UPDATE command to sequence the file.

update (noseq8: If you use the UPDATE command with a file that has been sequenced using the default values of XEDIT, you must use the NOSEQ8 option. Otherwise, the UPDATE command cannot process your input file. The command:

```
update sample (noseq8
```

tells UPDATE to use only the last five columns when it looks for sequence numbers.

Now, you can compile the new source file created by the UPDATE command.

UPDATE File

The UPDATE file usually has a file type of UPDATE. For convenience, you can give it the same file name as your source file.

The UPDATE file consists of two types of records:

- New or changed source statements
- Update control statements.

UPDATE Control Statements

Some update control statements are automatically generated by the editor, while others must be entered manually.

The editor generates the following three update control statements:

Statement	Type
./ I	Insert
./ D	Delete
./ R	Replace

The editor records these three control statements, along with the appropriate new or changed source statements, in the UPDATE file. Each update control statement also carries a time and date stamp in columns 52 through 71, reflecting when you created or changed the UPDATE file.

The layout of the update control statements is:

Columns	Contents
1-2	./
4	I, D, or R
6-13	Sequence number of source statement
24	$ (or other delimiter)
26-29	Starting statement number value, if this card applies to more than one statement in the source file.
31-33	Incrementing statement number value, if this card applies to more than one statement in the source file.

INSERT Statement: The INSERT statement precedes new records that you want to add to a source file. The INSERT statement tells the UPDATE command where to add the new records. For example, the lines:

```
./ I  1600
TEST2  TM   HOLIDAY,X'02'   HOLIDAY?
       BNO  VACATION        NOPE...VACATION
```

insert two lines of code, following the statement numbered 00001600, into the output file. The inserted lines are flagged with asterisks in columns 73 through 80, assuming this is an 80-character file (or the last eight columns if the logical record length is larger than 80). The INSERT statement also allows you to request that new statements be sequenced.

DELETE Statement: The DELETE statement tells the UPDATE command the records to delete from the source file. For example, the statement:

```
./ D  2500
```

deletes record 00002500 from the source file. The statements:

```
./ D  2500  2800
```

deletes all the statements from 2500 through 2800 from the source file.

REPLACE Statement: The REPLACE statement replaces one or more records in the source file. It precedes the new records you want to add. It is a combination of the DELETE and INSERT statements. For example, the lines

```
./ R 38000 38500
PLIST    DS    0D
         DC    CL8'TYPE'
         DC    CL8' '
         DC    CL8'FILE'
         DC    CL8'A1'
         DC    8X'FF'
```

replace the existing statements numbered 38000 through 38500 with the new lines of code. The new lines are not automatically resequenced.

The update facility does not automatically generate the following two update control statements. If you use them, use the editor to manually add them into the UPDATE file.

Statement	Type
./ S	Cause Resequencing
./ *	Comment

SEQUENCE Statement: The SEQUENCE statement tells the UPDATE command you want to number or renumber the records in a file. Sequence numbers are written in the last eight columns of the source file. For example, the statement:

```
./ S  1000
```

indicates that you want sequence numbering to be done in increments of 1000 with the first statement numbered 1000. The SEQUENCE statement is convenient if you want to apply updates to a file that does not already have sequence numbers. In this case, you may want to use the REP (replace) option of the UPDATE command, so that instead of creating a new file ($file name), you replace the original source file. For example:

```
update sample (rep
```

COMMENT Statement: Use the COMMENT statement to document the updates. These comments will appear in the update log file. For example, the line:

```
./ * Changes by John J. Programmer
```

is not processed by the UPDATE command when it creates the new source file, but it is written into the update log file.

Making Multiple Updates to a Source File Using the UPDATE Command

If you have several UPDATE files to apply to the same source, you may apply them in a series of UPDATE commands. For example, if you have updates named FICA UPDTUP1, FICA UPDTUP2, and FICA UPDTUP3 to apply to the source file FICA PLIOPT, you could do the following:

1. Update the source file with TEST1 UPDATE:

   ```
   update fica pliopt a fica updtup1
   ```

2. Update the source file produced by the previous command with the TEST2 UPDATE:

```
update $fica pliopt a fica updtup2
```

3. Update the new source file with TEST3:

```
update $$fica pliopt a fica updtup3
```

This final UPDATE command produces the file $$$FICA PLIOPT, which is now the fully updated source file. This method is cumbersome, however, particularly if you have many updates to apply. They must be applied in a particular order.

However, the UPDATE command provides a multilevel updating scheme that you can use to apply many updates at one time, in a specified order. There are two ways you can apply multilevel updates to a source file using the UPDATE command:

- Using a control file
- Using an auxiliary control file.

Using a Control File

A control file is actually a list. A control file, by convention, has a file type of CNTRL and a file name that is the same as the source input file. It does not contain any actual update control statements (INSERT, DELETE, and so on), but rather it indicates what UPDATE files should be applied, and in what order.

In the case of a multilevel update, all the UPDATE files must have the same file name as the source file. Therefore, only the file types need be specified in the control file to uniquely identify the UPDATE file. In fact, because all your UPDATE files specified in a control file must have file types beginning with the characters UPDT, you need only specify the unique part of the file type.

For example, to apply the three UPDATE files to FICA PLIOPT described earlier, you should create a file named FICA CNTRL. The control file for FICA PLIOPT, named FICA CNTRL, may typically look like the following:

```
TEXT MACS PLILIB
FICA3 UP3
FICA2 UP2
FICA1 UP1
```

The first noncommentary record in the control file must be a MACS record. A control file can contain multiple contiguous MACS records. The second field in this record must be “MACS”, and it may be followed by as many macro library names that will fit on the line. Every record in the control file must have an "update level identifier". In this example, the update level identifiers are TEXT on the MACS record, FICA1 for the UP1 record, and so on. The update level identifier may have a maximum of five characters.

The UPDATE command only uses the MACS record and the update level identifier under special circumstances. These are described later under “VMFASM EXEC Procedure” on page 82. For now, you only need to know that these things must be in a control file in order for the UPDATE command to execute properly.

Then, to update FICA PLIOPT, enter the UPDATE command as follows:

```
update fica pliopt (ctl
```

When you use the CTL option and you do not specify the name of a control file, the UPDATE command looks for a control file with the file type of CNTRL and a file name the same as the source file. From the control file, it reads the file types of the updates to be applied. In this example, the UPDATE command searches for the file FICA UPDTUP1 and if found, applies the updates; then UPDATE searches for FICA UPDTUP2, and applies those updates, if any. Last, it searches for FICA UPDTUP3, and applies those updates.

Notice that the updates are applied from the bottom of the control file, toward the top. This becomes important when an update is dependent on a previous update. For example, if you add some lines to a file in FICA UPDTUP1, then modify one of those lines in FICA UPDTUP2, it is important that UPDTUP1 was applied first.

Alternate Ways of Naming a Control File

The preceding example, showing FICA CNTRL and UPDTxxxx files, illustrates a naming scheme using the UPDATE command defaults. You can override the defaults for the control file's file name and file type.

If you name a control file GROUPA CNTRL, for example, you can specify the name of the control file on the UPDATE command line:

```
update fica pliopt a groupa cntrl (ctl
```

Using an Auxiliary Control File (AUX File)

The two levels of update processing shown so far may be adequate for your applications. There is, however, an additional level or step in the update structure that the z/VM procedures use and that you may want to use also.

These techniques may be useful when you have more than one set of updates to apply to a source program. For example, you may have two groups of programmers who are working on different sets of changes for the same source file. Each group may create several UPDATE files and have a unique control file. When you combine these changes, you could create one control file or you can use what are known as auxiliary control files.

The updating structure for auxiliary control files is based on conventions for assigning file names and file types. If a control file contains an entry that begins with the characters “AUX”, the UPDATE command assumes that the file "fn AUXnnnn" contains a list of file types, not UPDATE control statements. For example, if the file SAMPLE ASSEMBLE is being updated with a control file that contains the record:

```
TEST1 AUXLIST
```

Then SAMPLE AUXLIST does not contain UPDATE control statements. It contains entries indicating the **file types** of the UPDATE files, all of which must have the same file name, SAMPLE.

To see how this structure works, assume we have a source file, SAMPLE ASSEMBLE, and a control file, SAMPLE CNTRL. The file SAMPLE CNTRL contains the entries:

```
TEXT MACS
3676 AUXLIST
```

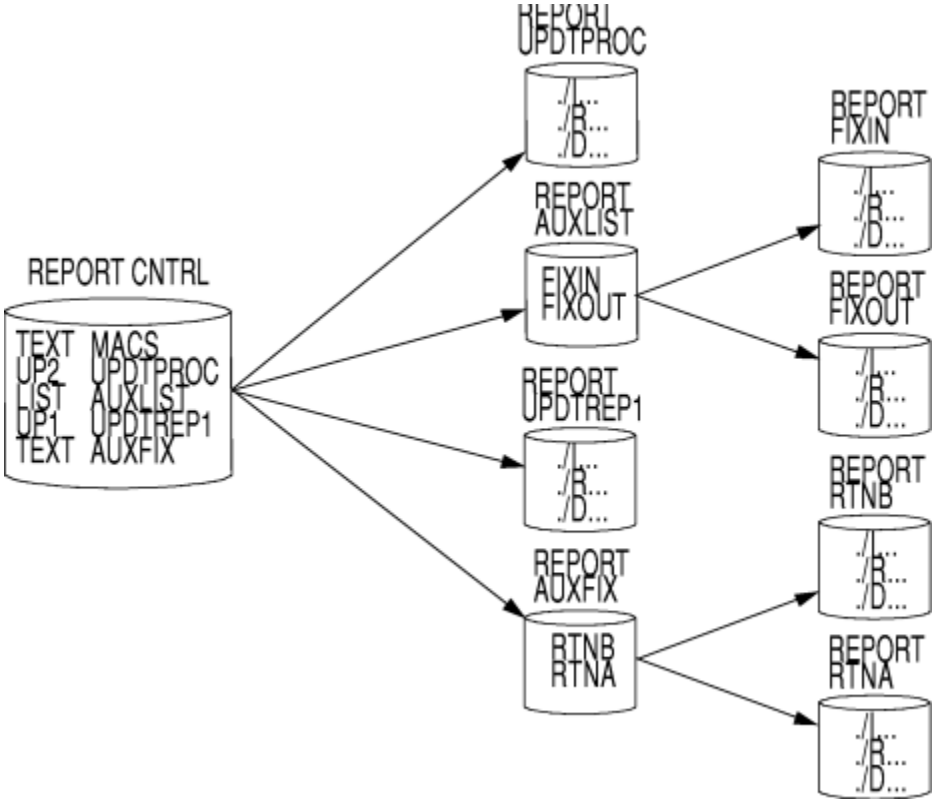
The file, SAMPLE AUXLIST may look like the following:

```
TEST1
FIXLOOP
BYPASS
```

The files:

```
SAMPLE TEST1
SAMPLE FIXLOOP
SAMPLE BYPASS
```

Contain UPDATE control statements (INSERT, DELETE, and so on) to be applied to the file SAMPLE ASSEMBLE. As with control file processing, the updates are applied from the bottom of the AUX file, so the updates in SAMPLE BYPASS are applied first, then the updates in SAMPLE FIXLOOP are applied, and so on. Figure 10 on page 80 shows an illustration of a set of UPDATE files.




```
update report assemble a (ctl
UPDATING 'REPORT ASSEMBLE A1' WITH 'REPORT RTNA A1'.
UPDATING WITH 'REPORT RTNB A1'.
UPDATING WITH 'REPORT UPDTREP1 A1'.
UPDATING WITH 'REPORT FIXOUT A1'.
UPDATING WITH 'REPORT FIXIN A1'.
UPDATING WITH 'REPORT UPDTRPOC A1'.
```

Figure 10. An Update with a Control File

Because the updating scheme uses only file types to uniquely identify UPDATE files, it is possible to use the same control file to update different source input files. For example, enter the following command when using the control file REPORT CNTRL shown in Figure 10 on page 80:

```
update fica pliopt a report cntrl (ctl
```

The UPDATE command begins searching for updates to apply to FICA PLIOPT, based on the entries in REPORT CNTRL. It searches for FICA AUXFIX, which may contain entries pointing to UPDATE files; then it searches for FICA UPDTREP1, and so on.

As long as all updates and auxiliary files associated with a source file have the same file name as the source file, the updates are uniquely identifiable. Therefore, the same control file can be used to update various source files. z/VM takes advantage of this capability in its own updating procedures. By maintaining strict naming conventions, updates to various CP and CMS modules are easily controlled and identified.

A control file may point to many AUX files in addition to many UPDT files. You can modify a control file when you want to control which updates are applied to a program. You may have several control files, and specify the name of the control file you want to use on the UPDATE command line. There is a lot of flexibility in the UPDATE command processing. You can implement procedures and conventions for your individual applications.

Making Multiple Updates to a Source File Using the XEDIT Command

Similar to the UPDATE command, you can use a control file and an auxiliary control file to apply many updates at one time using the XEDIT command with the CTL option.

Using a Control File

The XEDIT CTL option creates multiple updates to a source file. First, create a control file listing the updates to be applied to a source file. Initially, you might have only the MACS record and one UPDATE file type specified. For example, you can create a file called FICA CNTRL that contains:

```
TEXT MACS PLILIB
FICA1 UPDTUP1
```

Next, specify the control file name that you have created after the XEDIT CTL option. For example:

```
xedit fica pliopt (ctl fica
```

The editor searches for an UPDATE file called FICA UPDTUP1 and applies all updates contained in this file. If the UPDATE file does not exist, XEDIT creates a file called FICA UPDTUP1 which will contain all changes made to the source file during the editing session in addition to the required control statements.

If you wish to add another level of updates to your source file, insert a new UPDATE file type in your control file after the MACS record, for example:

```
TEXT MACS PLILIB
FICA2 UPDTUP2
FICA1 UPDTUP1
```

Then, XEDIT your source file again, specifying the CTL option, for example:

```
xedit fica pliopt (ctl fica
```

XEDIT applies all updates contained in FICA UPDTUP1 to the source file FICA PLIOPT. After the resulting file is displayed, any additional updates and the necessary control statements are automatically inserted in another UPDATE file called FICA UPDTUP2, consistent with control file processing from the bottom up.

Using an Auxiliary Control File (AUX File)

Auxiliary control files can also be used with XEDIT. You can make your control file point to AUX files that contain the file types of the actual UPDATE files, or you can combine AUX files and UPDATE files in a single control file. XEDIT begins applying updates from the bottom up in the control file and references the AUX files indicated. Any updates to the source file produced during the editing session are inserted in the topmost UPDATE file type specified in either the control file or in the last AUX file encountered using the 'bottom up' processing rule. More information about the XEDIT CTL option can be found in the *z/VM: XEDIT Commands and Macros Reference*.

Preferred Level Updating

There may exist more than one version of an update, each applicable to different versions of the same module. For example, you may need one version of an update for an unmodified base source module and another version of that update if a licensed program modified the modules. The AUX file used to update a particular module must then be selected based on whether or not a licensed program modifies that module. The AUX files listing the updates applicable to modules modified by a licensed program are called preferred AUX files because they must be used if they exist rather than the mutually exclusive updates applicable to unmodified modules. Using this preferred AUX file concept, every module in a component can be assembled using the one CNTRL file applicable to a user's configuration.

A single AUX file entry in a CNTRL file can specify more than one file type. The first file type indicates a file that UPDATE uses only on one condition: the files that the second and subsequent file types indicate do

not exist. If they do exist, this AUX file entry is ignored and no updating is done. The files that the second and subsequent file types indicate are preferred because UPDATE does not use the file that the first file type indicates. Usually, the preferred files appear later in the CNTRL file in a format that causes them to be used for updating.

UPDATE scans each CNTRL file entry until a preferred file type is found, until there are no more file types on the entry, or until a comment is found. (A character string less than four or more than eight characters is assumed to be a comment.)

VMFASM EXEC Procedure

If you are an assembler language programmer and you are using the UPDATE command to update source programs, you may want to use the VMFASM EXEC procedure. VMFASM is a z/VM update procedure. It invokes the UPDATE command and uses the ASSEMBLE command to assemble the updated source file.

There are other update procedures. The VMFHASM EXEC is an update procedure for Assembler H programs. The VMFHLASM EXEC is an update procedure for IBM High Level Assembler programs. See the *z/VM: VMSES/E Introduction and Reference* for more information.

If you are not an assembler language programmer, you may wish to create an exec similar to VMFASM that calls one of the language compilers to compile an updated source file, instead of calling the assembler.

When you use VMFASM, you specify the source file name, the file name of the control file, and optionally, parameters for the assembler. (The control file for VMFASM must have a file type of CNTRL). For example, if you use the file GENERAL CNTRL to update SAMPLE ASSEMBLE, enter:

```
vmfasm sample general
```

The VMFASM EXEC uses the MACS card and the update level identifiers in the control file. It reads the MACS card to determine which macro libraries (MACLIBs) should be searched by the assembler. Then VMFASM issues the GLOBAL MACLIB command specifying the MACLIBs you name on the MACS card.

VMFASM uses the update level identifier to name the output text file produced by the assembly. If the update level identifier of the most recent UPDATE file (the last one located and applied) is anything other than TEXT, the update level identifier is prefixed with the characters TXT to form the file type. For example, if the file GENERAL CNTRL contains the records:

```
TEXT MACS DMSGPI MYLIB OSMACRO
UP2 FIX2
UP1 FIX1
TEXT AUXLIST
```

and updates the file SAMPLE ASSEMBLE, then:

- If the file SAMPLE UPDTFIX2 is found and the updates applied, VMFASM names the output text deck SAMPLE TXTUP2.
- If the file SAMPLE UPDTFIX1 is found and the updates applied but no SAMPLE UPDTFIX2 is found, the text deck is named SAMPLE TXTUP1.
- If the file SAMPLE AUXLIST is found but no SAMPLE UPDTFIX1 or SAMPLE UPDTFIX2 files are found, the text deck is named SAMPLE TEXT.
- If no files are found, the update level identifier on the MACS card is used and the text deck is named SAMPLE TEXT.

The new fn TEXT or fn TXTxxxxx resides on the A-disk. Because the UPDATE command works from the bottom of a control file toward the top, it is logical that the text file name be taken from the identifier of the last update applied.

The VMFASM EXEC does not produce an updated source file, but leaves the original source intact.

VMFASM produces two output files:

- A printed output listing that shows update activity

- The text file that contains the update log as well as the actual object code.

If you use the CMS LOAD command to load a text file produced by VMFASM, records from the update log are flagged as not valid, but the LOAD operation is not impaired.

Making Updates to Execs and Macros Using the EXECUPDT Command

If you wish to use the update facility to track changes to execs or macros written for REXX, you need to use the EXECUPDT command. The EXECUPDT command applies updates to an exec source file (using the UPDATE command) and removes the sequence numbers from the updated file to produce an executable version of the file. Using EXECUPDT is very similar to using the VMFASM EXEC to apply updates to an assembler language source and to assemble it.

Source files for the EXECUPDT command are fixed-length files with sequence numbers just like those for assembler language or COBOL. Note, the default record length for these files is 80 characters but files could have record lengths up to 255 characters. The file type of the exec source file has a '$' prefixed to the normal file type. For example, SAMPLE $EXEC could be the source for an exec procedure and READY $XEDIT could be a source file for an XEDIT macro.

Updates to the exec source are created using XEDIT in the same manner as updates to programs in other languages. To apply the updates to the source, use the EXECUPDT command. For a single level update, enter:

```
execupdt sample exec
```

Note that the '$' in the file type is not included in the file type specified on the EXECUPDT command. To do a multilevel update, you may use the CTL option of EXECUPDT. For example:

```
execupdt sample exec (ctl general
```

Writing Your Own Exec to Invoke the UPDATE Command (The STK Option)

If you are interested in writing your own EXEC procedure to invoke the UPDATE command, you may wish to use the STK option. The STK (stack) option is valid only with the CTL option and is meaningful only when the UPDATE command is invoked within an EXEC procedure.

When the STK option is specified, UPDATE stacks the following data lines in the console stack:

```
first line:  * update level identifier
second line: * library list from MACS record
```

The update level identifier is the identifier of the most recent update that was found and applied. Comments may be specified on MACS records by means of an asterisk (*). Any information beyond the * is treated as a comment. The comments are not passed on when you specify the STK option.

The following REXX program issues the UPDATE command and then the ASSEMBLE command:

```
/* Sample REXX program to update               */
/* and assemble a source program               */
address command
parse upper arg fname cntrl '(' options
'UPDATE' fname 'ASSEMBLE *' cntrl 'CNTRL * ( CTL STK'
parse upper pull star txtype .
parse upper pull star maclibs
'GLOBAL MACLIB' maclibs
if txtype ¬= 'TEXT' then
  'FILEDEF TEXT DISK' fname 'TXT'txtype 'A1'
'ASSEMBLE $'fname '(' options
'ERASE $'fname 'ASSEMBLE'
```

If the EXEC that you use is named UPASM EXEC, it is invoked with the line:

```
upasm fica fica (print noxref
```

and the file FICA CNTRL contains:

```
MAC MACS DMSGPI OSMACRO MYTEST  * Comments
FIX1  UPDTFIX
LIST  AUXLIST
```

then the REXX exec executes the following:

```
UPDATE FICA ASSEMBLE * FICA CNTRL * (STK CTL
GLOBAL MACLIB DMSGPI OSMACRO MYTEST
FILEDEF TEXT DISK FICA TXTFIX1 A1
ASSEMBLE $FICA (PRINT NOXREF
ERASE $FICA ASSEMBLE
```

The previous example assumes that the UPDATE file FICA UPDTFIX was found and applied.

Example of Updating a FORTRAN Source File

The following is an example of updating a FORTRAN source file. Enter:

```
type testprog fortran
```

The result should look like this:

```
      PROGRAM MYPROG                                                TES00010
      CHARACTER*8 F,S                                               TES00020
      WRITE (6,5)                                                   TES00030
      READ (5,2) F                                                  TES00040
      WRITE (6,10)                                                  TES00050
      READ (5,2) S                                                  TES00060
      WRITE (6,15) F,S                                              TES00070
 2    FORMAT (A8)                                                   TES00080
 5    FORMAT (' ENTER YOUR FIRST NAME.')                            TES00090
 10   FORMAT (' AND NOW YOUR LAST NAME.')                           TES00100
 15   FORMAT (' WELCOME TO CMS, ',A8,1X,A8)                         TES00110
      STOP                                                          TES00120
      END                                                           TES00130
```

There are sequence numbers in columns 73 through 80, which were automatically generated by the editor. Each sequence number is prefixed with the letters TES, the first three letters of the file name. The editor generated the sequence numbers in this form because the default file characteristics were in effect:

- TRUNC was set to 72, allowing serialization in columns 73 to 80.
- SERIAL was set to ON 10 10, meaning that:
  - The first three positions of the sequence number would be filled with the first three characters of the file name.
  - The numeric portion of the sequence number (columns 76 through 80) would begin with 10.
  - Each subsequent sequence number would be incremented by 10.

To use the UPDATE option of the editor, you have to convert the sequence numbers to all numerics—that is, all eight characters of the sequence number must be used. The editor makes this task very simple. First, bring the program into the editor by entering:

```
x testprog fortran (noprof
```

Now enter:

```
serial all 10 10
file
```

SERIAL ALL means that all eight characters of the sequence field are used for numeric sequencing. Type the file on the terminal again using the TYPE command. The file should look like this:

```
      PROGRAM MYPROG                                                    00000010
      CHARACTER*8 F,S                                                   00000020
      WRITE (6,5)                                                       00000030
      READ (5,2) F                                                      00000040
      WRITE (6,10)                                                      00000050
      READ (5,2) S                                                      00000060
      WRITE (6,15) F,S                                                  00000070
2     FORMAT (A8)                                                       00000080
5     FORMAT (' ENTER YOUR FIRST NAME.')                                00000090
10    FORMAT (' AND NOW YOUR LAST NAME.')                               00000100
15    FORMAT (' WELCOME TO CMS, ',A8,1X,A8)                             00000110
      STOP                                                              00000120
      END                                                               00000130
```

Now the sequence numbers are all numeric.

Suppose you want to make the following changes:

- Add a line between the second and third record of the file (that is, between the CHARACTER*8 statement and the WRITE (6,5) statement. This line contains a comment line giving the programmer's name.
- Move line 9 to follow line 10.

To make these changes, follow these steps:

1. Call the editor with the update option, by entering:

   ```
   x testprog fortran (update noprof
   ```

   Notice that the file type is now UPDATE.
2. Position the cursor in the prefix area of line 2 (the CHARACTER*8 statement), and enter:

   ```
   ==a==
   ```

   This adds a new line. The cursor is now at the beginning of the new line.
3. Because the comment must begin in column 1, you can now type the comment line:

   ```
   c author. sam jones.
   ```
4. Enter the DOWN 5 command. Now move the cursor to the 10th line, which reads:

   ```
   5     FORMAT (' ENTER YOUR FIRST NAME.')
   ```

   and enter:

   ```
   m====
   ```

   You need a target to move the line to, so move the cursor to the next line, which reads:

   ```
   10    FORMAT (' AND NOW YOUR LAST NAME.')
   ```

   and enter:

   ```
   f====
   ```

   The result is that lines 9 and 10 have swapped position.
5. Now, close out the editing session by entering:

   ```
   file
   ```

If you now type the original program, TESTPROG FORTRAN, you will see that none of the changes you made have taken place in the source file. This is because the update option has created a new file

called TESTPROG UPDATE, which contains the changes you made, together with the control statements necessary to implement the changes again. Now type the TESTPROG UPDATE file. It should look like this:

```
./ I 00000020          $ 25 5                          03/08/01 11:24:11
C AUTHOR. SAM JONES
./ D 00000090                                          03/08/01 11:24:11
./ I 00000100          $ 105 5                         03/08/01 11:24:11
5     FORMAT (' ENTER YOUR FIRST NAME.')
      00000090
```

Note: The date and time stamp values reflect the time you entered the FILE subcommand.

Now we will use the UPDATE command to update the source. Enter:

```
update testprog fortran
```

This updates the source and creates a new file called $TESTPRO FORTRAN and an update log called TESPTOR UPDLOG. The source file $TESTPRO FORTRAN should look like this:

```
       PROGRAM MYPROG                                                    00000010
       CHARACTER*8 F,S                                                   00000020
 C AUTHOR. SAM JONES.                                                    ********
       WRITE (6,5)                                                       00000030
       READ (5,2) F                                                      00000040
       WRITE (6,10)                                                      00000050
       READ (5,2) S                                                      00000060
       WRITE (6,15) F,S                                                  00000070
 2     FORMAT (A8)                                                       00000080
 10    FORMAT (' AND NOW YOUR LAST NAME.')                               00000100
 5     FORMAT (' ENTER YOUR FIRST NAME.')                                ********
 15    FORMAT (' WELCOME TO CMS, ',A8,1X,A8)                             00000110
       STOP                                                              00000120
       END                                                               00000130
```

The update log file TESTPROG UPDLOG should look like this:

```
1UPDATING 'TESTPROG FORTRAN    A1' WITH 'TESTPROG UPDATE   A1'
UPDATE LOG -- PAGE       1 0       ./ I 00000020          $ 25
5                        03/08/01 11:24:11 INSERTING...    C AUTHOR.
SAM JONES.
        ********
        ./ D 00000090
03/08/01 11:24:11 DELETING...     5     FORMAT (' ENTER YOUR FIRST
NAME.')
          00000090
        ./ I 00000100          $ 105 5
03/08/01 11:24:11 INSERTING...    5     FORMAT (' ENTER YOUR FIRST
NAME.')
        ********
```

Chapter 9. Building and Using Dynamic Link Libraries (DLLs)

A dynamic link library (DLL) is a collection of one or more functions or variables gathered in a load module and executable or accessible from a separate application load module. The term derives from the fact that the connection or link between the application that uses the DLL and the DLL functions or variables is made dynamically while the application is executing rather than statically when the application is built. You can, therefore, call a function or use a variable in a load module other than the one that contains the definition. You can use DLLs both implicitly (load-on-call) and explicitly.

When an application implicitly calls a DLL function or references an imported variable, the DLL is implicitly loaded. This load-on-call use of a DLL is essentially transparent to the application code. All the connections for the references to the definitions that belong to the DLL are established when the DLL is first called or referenced.

The use of DLLs can also be explicitly controlled by the application code at the source level. The application uses explicit source-level calls to one or more execution-time services. The connections for the reference and the definition are made at run time.

The information in this chapter introduces DLL concepts and describes how to build DLLs and applications that use DLLs. For additional information about compiler options for DLLs or about prelinking applications, see the *XL C/C++ for z/VM: User's Guide*.

DLL Concepts and Terms

This section presents key concepts for understanding DLLs.

DLLs and DLL Applications

A **DLL** is a load module that exports function or variable definitions to other DLLs or DLL applications.

A **DLL application** is an application that references imported functions or imported variables. A DLL can also import functions and variables from other DLLs.

Imported and Exported Functions and Variables

Imported functions and variables are not defined in the load module where the reference to them is made.

Non-imported functions and variables are defined in the same load module where a reference to them is made.

Exported functions or variables are defined in one load module and can be referenced from another load module. If an exported function or variable is also referenced within the load module where it is defined, the exported function or variable is also non-imported.

DLL Code and Non-DLL Code

DLL code and **non-DLL code** are types of object code. The difference between them is the way they reference functions and external variables. In DLL code, special code sequences are always generated by the compiler for referencing functions, external variables, and using function pointers. With these code sequences, a DLL application can reference imported functions and imported variables from a DLL as easily as it can non-imported ones.

The object code generated by the C/C++ compiler with the DLL compiler option is DLL code.

Other types of object code are classified as non-DLL code.

Function and Variable Descriptors

A **function descriptor** is an internal control block that contains the function address.

A **variable descriptor** is an internal control block that contains the variable address.

Definition Side-Deck

When you build a DLL, a definition side-deck is created on output by the prelinker. It is a directive file that contains an **IMPORT** control statement for each function and variable exported by that DLL. You must include this definition side-deck when you prelink a DLL application that imports any functions or variables from the DLL.

Building a DLL or a DLL Application

Building a DLL or a DLL application is similar to creating any C application. It involves three steps:

1. Compiling
2. Prelinking

   Note: Although some C applications may not require prelinking, the prelink step is mandatory when you are building a DLL or a DLL application.
3. Building

To build a DLL or a DLL application, you must pay special attention to the compile and the prelink steps.

The following sections describe how to build simple DLLs or DLL applications. "Building a Complex DLL or DLL Application" on page 90 discusses how to build a complex DLL or DLL application. A complex DLL could be any of the following:

- A DLL that consists partially of C source files and C++ source files, and partially of other types of source files, such as an assembler source file.
- A DLL that contains C source files (compiled with the DLL compiler option) that must also be linked with additional object text decks, such as C text decks compiled without DLL. For example, to statically link your DLL to a TEXT library so that you can package them together, you need to create a complex DLL.
- A DLL that is intended to support both DLL and non-DLL applications.

Building a Simple C DLL

You can build a simple C DLL from C source files that contain some defined functions or variables with external linkage that you want to export to the users of the DLL.

Each function that you want to export from the DLL must be an external function.

To build a simple C DLL:

1. Write code using the `#pragma export` directive to export specific external functions and variables:

```
#pragma export(goo)
int foo() {
  ...
}
int goo() {
  ...
}
int kpItHdn() {
  ...
}
  ...
#pragma export(hooVar)
#pragma export(kooVar)
int hooVar;
int kooVar;
int kpItHdnVar;
```

For the above example, the functions `foo()` and `goo()` and the variables **hooVar** and **kooVar** are exported. The function `kpItHdn()` and the variable **kpItHdnVar** are not exported.

Note: If you want to export *all* defined functions and variables with external linkage in the compilation unit to the users of the DLL, compile with the EXPORTALL compile option. This means that all defined functions and variables with external linkage will be accessible from this DLL and by all users of this DLL.

Using EXPORTALL means that you do not need to include `#pragma export(...)` in your code.

2. Compile with the DLL compiler option. For example, when using c89:

```
c89 ... -Wc,dll foogoo.c ...
```

This option instructs the compiler to generate special code when calling functions and referencing external variables. For a simple DLL that does not reference any imported functions or imported variables from other DLLs, specifying the DLL compiler option is not mandatory but it is recommended. Always compiling a simple DLL as DLL code eliminates the potential compatibility problems that may occur when linking DLL code with non-DLL code. See "Building a Complex DLL or DLL Application" on page 90 for more information on compatibility issues.

3. Prelink with the DLL prelinker option. For example, when using c89:

```
c89 ... -Wb,p,dll ... foogoo.o ...
```

No special text decks other than those for creating the DLL are required. The prelinker automatically creates a definition side-deck. For the users of this DLL, the definition side-deck describes the functions and the variables that can be imported by DLL applications. A DLL provider must provide this side deck to the users of the DLL, and the users must include it when they prelink a DLL application.

When prelinking the TEXT file for the above source files, the prelinker generates the following definition side-deck:

```
IMPORT CODE 'FOOGOO'   goo
IMPORT DATA 'FOOGOO'   hooVar
IMPORT DATA 'FOOGOO'   kooVar
```

FOOGOO is the name or alias of the load module for the DLL. The prelinker puts out the DLL name in single quotation marks, and leaves enough extra blanks such that you can change FOOGOO to an 8-character name without moving the function or variable name.

You can edit the definition side-deck to remove any functions or variables that you do not want to export. For instance, in the above example, if you do not want to expose `goo()`, remove the control statement `IMPORT CODE 'FOOGOO' goo` from the definition side-deck.

Note: You should also provide a header file containing the prototypes for exported functions and `extern` variable declarations for exported variables. DLL application writers can then include this header file in their source files that reference functions or variables in the DLL.

4. Build the DLL as you would any C application. For example, when using c89:

```
c89 -o foogoo ... foogoo.o ...
```

Building a Simple C DLL Application

A simple C DLL application contains C source files. Some of the files contain references to imported functions or imported variables.

To use a load-on-call DLL in your simple C DLL application:

1. Compile with the DLL compile option. It instructs the compiler to generate special code when calling functions and referencing external variables.

2. Prelink. Include the definition side-deck from the DLL provider in the set of text decks to prelink. The prelinker uses the definition side-deck to resolve references to functions and variables defined in the DLL. If you are referencing multiple DLLs, you must include multiple definition side-decks.

   Note: You must use an `INCLUDE` statement for each definition side-deck unless the DLL application uses only explicit DLL references. The automatic library call (autocall) process will not resolve these references if they are not explicit.

3. Build the DLL application as you would any C application.

See Figure 11 on page 93 for a summary of the processing steps required for a DLL application (and related DLLs).

Building a Complex DLL or DLL Application

A key characteristic of a complex DLL or DLL application is that it is created by linking DLL code with non-DLL code. Non-DLL code may be used because some of the source files are not written in C, or because some of the C source files must be compiled with the `NODLL` compiler option (refer to "Rules for Compiling DLL Code Versus Non-DLL Code" on page 90). For example, DLL application writers may want to statically link their applications to a TEXT library that shipped earlier.

These are the general steps for creating a complex DLL or DLL application:

1. Determine the source files that must be compiled as non-DLL code and assume that the rest of the source files will be compiled as DLL code.
2. Make sure that all the source files meet all the DLL rules. If there are violations, follow the corrective steps described in "Rules for Modifying DLL Source" on page 91.
3. Compile the source files to produce DLL code and non-DLL code as determined in the previous steps.
4. Prelink. Follow the steps for a simple DLL or DLL application.
5. Build the DLL or DLL application as you would any C application.

Rules for Compiling DLL Code Versus Non-DLL Code

To create a complex DLL or DLL application, you must decide which source files should be compiled as DLL code and which should be compiled as non-DLL code. The following is a list of the rules for making that decision:

1. A source file that contains a function call through a pointer that may point to either a function or function descriptor must be compiled as non-DLL code. For example, the `qsort()` routine in Figure 19 on page 103 must be compiled as non-DLL code, because `qsort()` gets a pointer to a function to call. Because that pointer can be to a function compiled as either DLL code or non-DLL code, the `qsort()` routine emitted by the compiler cannot tell whether the routine to invoke is DLL or non-DLL code. This means that `qsort()` must be non-DLL. For additional information about function pointers, see "Pointer Assignment" on page 98.
2. A source file that calls imported functions or imported variables by name must be compiled as DLL code.
3. A source file that contains a comparison of function pointers that may be DLL function pointers must be compiled as DLL code.

   The comparisons shown in "Function Pointer Comparison in Non-DLL Code" on page 104 are undefined. To obtain valid comparisons, you must compile the same source files as DLL code.

4. A source file that may pass a function pointer to DLL code through a parameter or a return value must be compiled as DLL code.

   If the `qsort()` routine in Figure 19 on page 103 is compiled as DLL code instead of non-DLL code, non-DLL applications can no longer call it. To be able to call the DLL code version of `qsort()`, the original non-DLL application must be recompiled as DLL code.

5. A source file that may pass a function pointer to DLL code by linking the references to the definition of an external variable must be compiled as DLL code.

In the following examples, if source file 1 has been compiled as DLL code, source file 2 must be compiled as DLL code as well.

a. File 1:

```
void main(void) {
  goo();           /* initialize fp to be hello function */
  fp();            /* call hello function                */
}
```

b. File 2:

```
extern void (*fp)(void);
void hello(void) {
  printf("hello\n");
}
void goo(void) {
  fp = hello;
}
```

If you do not use the DLL compiler option, **fp** would contain the address of **hello**. File 1 would expect **fp** to contain a function descriptor for **fp**, and the execution of **fp** would abend.

You must comply with all the rules described above when creating a complex DLL or DLL application. The prelinker flags violations of rule "2" on page 90 as an error, but the compiler and prelinker do not indicate any other rule violations.

Note: A DLL or DLL application that does not comply with these rules may produce undefined run-time behavior. For a detailed explanation of incompatibilities between DLL and non-DLL code, see "Compatibility Issues between DLL and Non-DLL Code" on page 96.

To comply with the DLL rules, you should also be aware of the following:

1. A C source file will be compiled as DLL code if the `DLL` compiler option is on.
2. A C source file will be compiled as non-DLL code if the `NODLL` compiler option is on.
3. Other types of source files can be compiled only as non-DLL code.

Rules for Modifying DLL Source

Sometimes, source files of a complex DLL or DLL application do not simultaneously meet all the DLL rules. For example,

1. If a C++ source file that is automatically compiled as DLL code contains a function call through a pointer that may be either a DLL function pointer or a non-DLL function pointer, it must be compiled as non-DLL code by DLL rule "1" on page 90. For example, a user writes the `qsort()` routine in Figure 19 on page 103.
2. A C source file that contains the `qsort()` routine (and therefore must be compiled as non-DLL code) also contains the source for a DLL application that must be compiled as DLL code.

When these situations occur, you can use the following methods to solve the problem:

1. Rewrite the source in C. Only the C source can be compiled as either DLL or non-DLL code. For example "1" on page 91 above, rewrite the `qsort()` routine in C.
2. Split a source file into two, so that one of the new files can be compiled as DLL code and the other can be compiled as non-DLL code. For example "2" on page 91 above, split the source file into two so that one contains the `qsort()` routine while the other contains a DLL application.

   Note: Sometimes you might have to split a function into two functions before you can successfully split the file.

Summary Example: Creating and Using DLLs

Figure 11 on page 93 summarizes the use of DLLs for both the DLL provider and for the writer of applications that use them. In this example, application ABC is referencing functions and variables from

two DLLs, XYZ and PQR. The connection between DLL preparation and application preparation is shown. Each DLL shown contains a single compilation unit. The same general scheme applies for DLLs composed of multiple compilation units, except that they have multiple compiles and a single prelink for each DLL. For simplicity, this example assumes that ABC exports variables or functions and that XYZ and PQR do not use other DLLs.

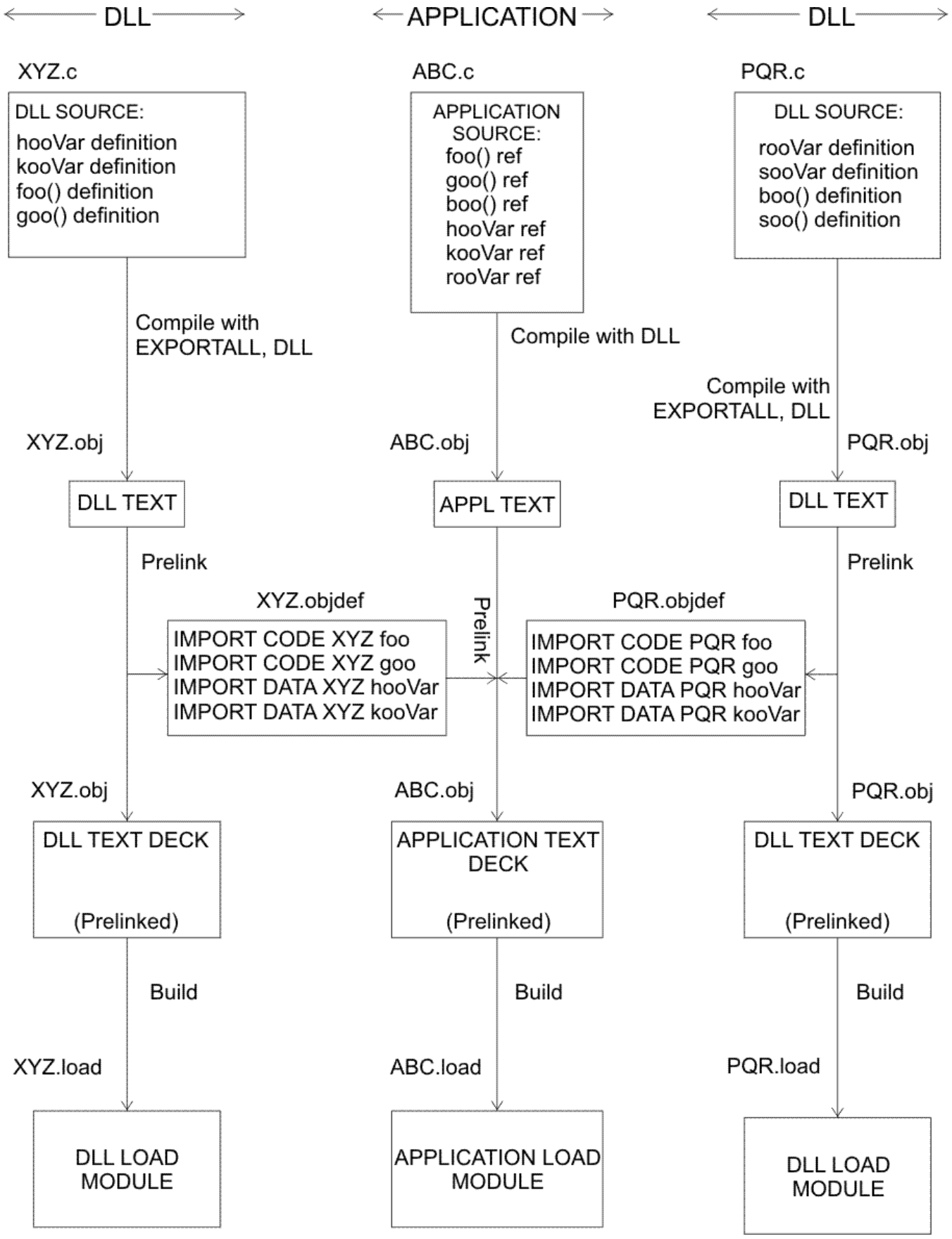


Figure 11. Summary of DLL and DLL Application Preparation and Usage

Managing the Use of DLLs when Running DLL Applications

This section describes how the C/C++ compiler manages loading and freeing DLLs when you run a DLL application.

Loading DLLs

There are four ways to implicitly load a DLL. They are:

1. Statically initializing a variable pointer to the address of an exported DLL variable
2. Calling an exported function
3. Referencing (using, modifying, or taking the address of) an exported variable
4. Calling through a function pointer that points to an exported function

In the first case, the DLL is loaded when writable static is initialized and (in C++) constructors are run before `main()` is invoked. In the other cases, the DLL is loaded at the time of the implicit call.

Note: In C++, constructors are run once on initial load and destructors are run once.

When a DLL is called explicitly, writable static is initialized and (in C++) constructors are run when the first call to `dllload()` is made. For more information on the library functions that make explicit calls to DLL services, see "Explicitly Calling a DLL" on page 107 and the *XL C/C++ for z/VM: Runtime Library Reference*.

When you are running with Language Environment, you can load DLLs from the OpenExtensions byte file system (BFS) as well as from accessed CMS minidisks and SFS directories. In most cases when the Language Environment run-time library attempts to load the DLL, it tries to load first from the BFS and then, if it does not find the DLL, from the CMS search order.

The exception to the above case is if the DLL name is unambiguous, or specifically a BFS path ID or CMS file ID. For example, if a DLL name starts with two slashes (//), the run-time library looks only in the accessed CMS search order. If the name starts with a single slash, the library looks only in the BFS.

When the run-time library attempts to load a DLL from the BFS, it will look in the directories specified by the `LIBPATH` environment variable. If `LIBPATH` is not specified, the current working directory is searched.

Sharing DLLs

DLLs are shared at the enclave level (as defined by Language Environment). A referenced DLL is loaded only once per enclave, and only one copy of the writable static is created or maintained per DLL per enclave. Thus, a single copy of a DLL serves all modules in a given enclave regardless of whether the DLL is loaded implicitly (through a reference to a function or variable) or explicitly (through `dllload()`). You can simultaneously access a given DLL within a given enclave both implicitly and by explicit execution-time services.

All accesses to a given variable in a given DLL in a given enclave refer to the only copy of that variable. All accesses to a given function in a given DLL in a given enclave refer to the only copy of that function.

Although only one copy of a DLL is maintained per enclave, multiple logical loads are counted and used to determine when the DLL (including its writable static area) is actually deleted. For a given DLL in a given enclave, there is one logical load of it for the first implicit reference to one of its variables or functions from a given load module, and one logical load for each explicit `dllload()` request.

DLLs are not shared in a nested enclave environment. Only the enclave that loaded the DLL can access functions and variables. For example, if **program A** loads a DLL and makes a system call to **program B**, **program B** cannot access any of the DLL variables or functions loaded by **program A**. **Program B** can access the DLL variables and functions by loading its own copy of the DLL.

Freeing DLLs

Explicitly-loaded DLLs may be freed with a `dllfree()` request. This request is optional because the DLLs are automatically deleted by the run-time library at enclave termination.

Implicitly-loaded DLLs cannot be deleted from the DLL application code. They are deleted by the run-time library at enclave termination.

DLL Restrictions

Consider the following restrictions when creating DLLs and DLL applications:

1. CEESTART must exist in the load module.
2. The AMODE of a DLL application must be the same as the AMODE of the DLL that it calls.
3. DLL facilities are not available under the SPC environment.
4. DLL facilities are not available to application programs with `main()` written in PL/I and that dynamically call C/C++ functions.
5. DLL facilities are not available to application programs written in COBOL and that dynamically call C/C++ functions.
6. You cannot implicitly load a DLL while running C++ static destructors.
7. You cannot use DLL static constructors to create new threads. This restriction is to prevent the creation of a DLL loop, where one DLL loads another, which then tries to load the first. In a single thread, if you try to create a DLL loop, you will get an error message. However, Language Environment supports DLLs in a multithreaded environment. If a DLL static constructor could create threads, there might be a case where DLL A loaded DLL B, whose static constructor created a new thread that tried to load DLL A before the static constructor for DLL B completed. The run-time library would not recognize this as a DLL loop. For this reason, DLL static constructors cannot create new threads.
8. You cannot use the functions `set_new_handler()` or `set_unexpected()` in a DLL if the DLL application is expected to invoke the new handler or unexpected function routines.
9. Be very careful when you use the explicit DLL functions in a multithreaded environment. Make sure you avoid any situation where one thread frees a DLL while another thread calls any of the DLL functions. Such a situation happens, for example, when a `main()` function uses `dllload()` to load a DLL, then creates a thread that uses the `ftw()` function. The `ftw()` target function routine is in the DLL. If the `main()` function uses `dllfree()` to free the DLL, but the created thread uses `ftw()` at any point, you will get an abend. There are many scenarios similar to this, all of which can be avoided if you do either of the following:
   - Do not free any DLLs by using `dllfree()`. (Language Environment will free them when the enclave is terminated.)
   - Have the `main()` function call `dllfree()` only after all threads have been terminated.

Performance Considerations

This section contains some hints on using DLLs efficiently.

- **Static Initialization in DLL code**

  Do not use the static initializations that require a DLL to be loaded if the code actually using the DLL may not be executed.

  The static initialization of an external or static pointer variable to the address of an imported variable implies loading a DLL. Static initialization occurs when a DLL application begins executing, before C++ static constructors are run and before the execution of the main function.

  A DLL needed for the static initialization is always loaded at initialization time, even if the code using those pointers may not be executed later. To minimize unnecessary startup time, use a run-time initialization so that the DLL is loaded only if it is actually used.

- **Group external variables**

  Group all external variables into one external structure.

- **POSIX Run-time Option and Loading DLLs**

When `POSIX(ON)` is used, Language Environment always tries to load DLLs from the BFS first, then from the accessed CMS search order. If the DLLs actually reside on CMS minidisks or SFS directories, this results in unnecessary load attempts. If an application must run with CMS DLLs, you can put two slashes (//) in front of the CMS DLL names before the application is prelinked, to make the DLL names in the side-decks unambiguous. If your application uses a DLL in the BFS, you can make the DLL name unambigious by putting a period and a slash (./) in front of it before you prelink.

- In C, do not compile with the `DLL` compiler option if the compilation unit does not use imported symbols.

Compatibility Issues between DLL and Non-DLL Code

This section describes the differences between DLL code and non-DLL code, and discusses the related compatibility issues for linking them to create complex DLLs.

Referencing Functions and External Variables

Table 5 on page 96, Figure 12 on page 97, and Figure 13 on page 98 show the differences between DLL and non-DLL code in referencing functions and variables. Undefined references fail during the link step.

Table 5. Referencing Functions and External Variables

	DLL	**Non-DLL**
Imported Functions	A function descriptor is created by the prelinker. The descriptor is in the writable static area (WSA) and contains the address of the function and the address of the writable static area associated with that function. The function address is resolved when the DLL is loaded. 1	Explicit reference is undefined 5
Non-imported Functions	Also called by the function descriptor, but the function address is resolved at load time. 3	Directly, by function address 7
Imported Variables	A variable descriptor is created in the WSA by the prelinker. It contains addressing information for accessing an imported variable. The address is resolved when the DLL is loaded. 2	Undefined 6
Non-imported Variables	Direct access 4	Direct access 8

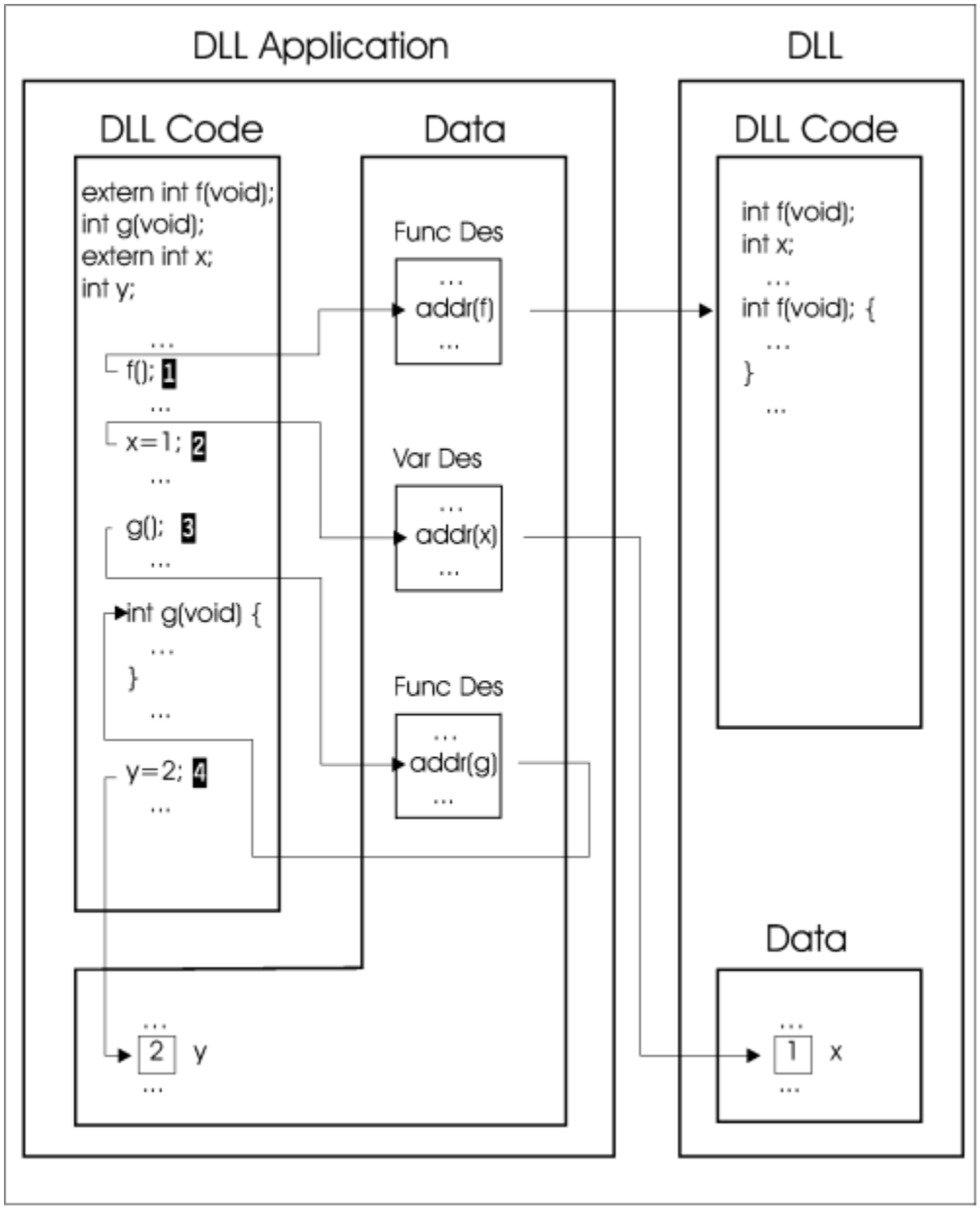


Figure 12. Referencing Functions and External Variables in DLL code

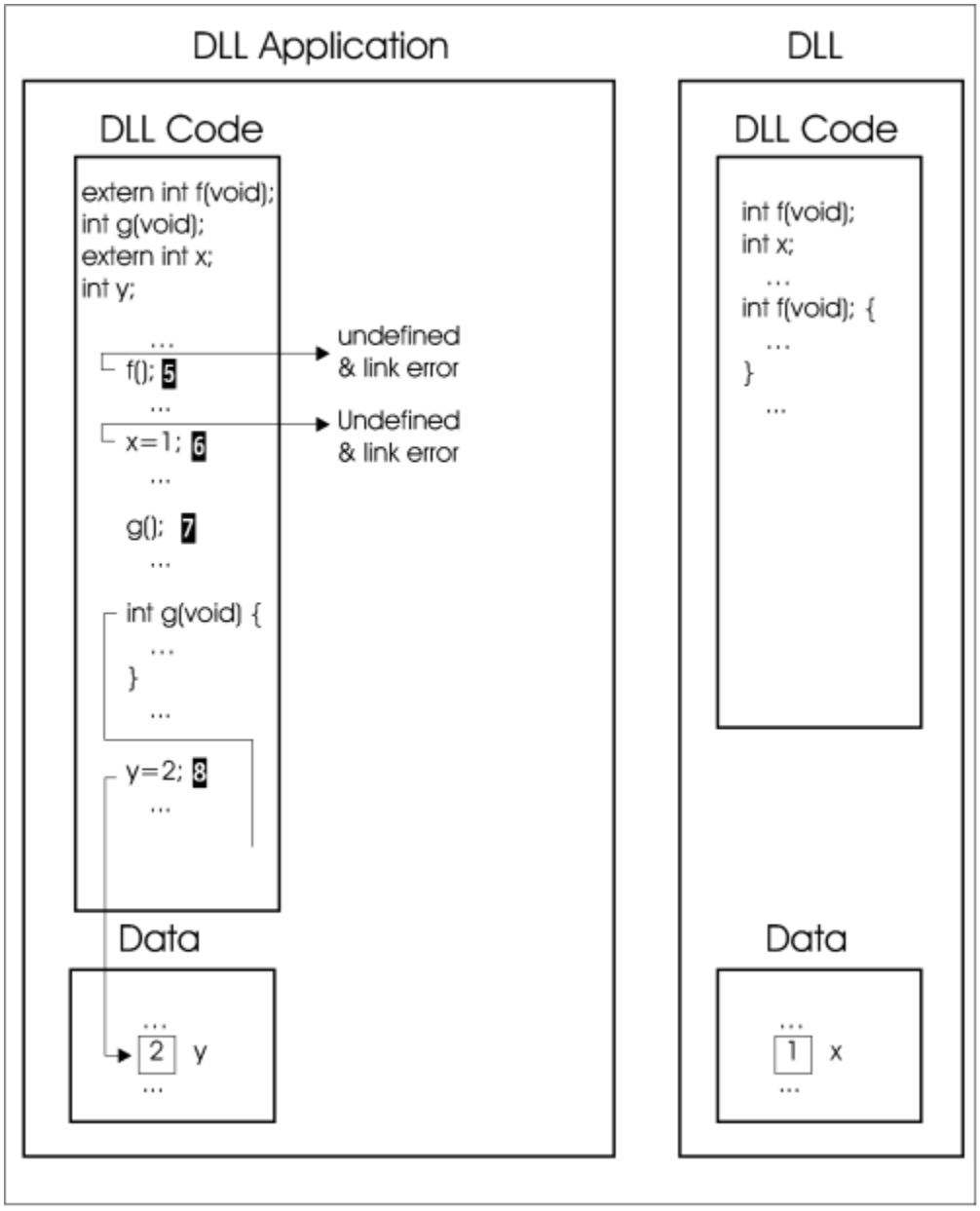


Figure 13. Reference of Functions and External Variables in non-DLL code

Pointer Assignment

External Variable Pointers

If you assign the address of an imported variable to a variable pointer in non-DLL code, any reference to that pointer will fail.

In DLL code and non-DLL code, the absolute address of a variable is assigned to a variable pointer.

A valid variable pointer always points to the variable itself and causes no compatibility problems.

Function Pointers

If you assign the address of an imported function to a pointer in non-DLL code, the link step will fail. In non-DLL code, the absolute address of a non-imported function is assigned to a function pointer. In DLL code, the address of a function descriptor is assigned to a function pointer.

In a complex DLL or DLL application, a DLL function pointer may be passed to non-DLL code and a non-DLL function pointer may be passed to DLL code. (A function pointer can be passed from DLL code to non-DLL code, or vice versa, by a parameter, a return value, or an external variable through the link step.) Thus, in a complex DLL or DLL application, a function pointer may point either to a descriptor or to a function entry, depending on whether the function pointer originates in DLL or non-DLL code. The different ways of de-referencing a function pointer causes the compatibility problem in linking DLL code with non-DLL code.

In Figure 14 on page 99, 1 assigns the address of the descriptor for the imported function `f` to `fp`. 2 assigns the address of the imported variable x to xp. 3 assigns the address of the descriptor for the non-imported function g to gp. 4 assigns the address of the non-imported variable y to yp.

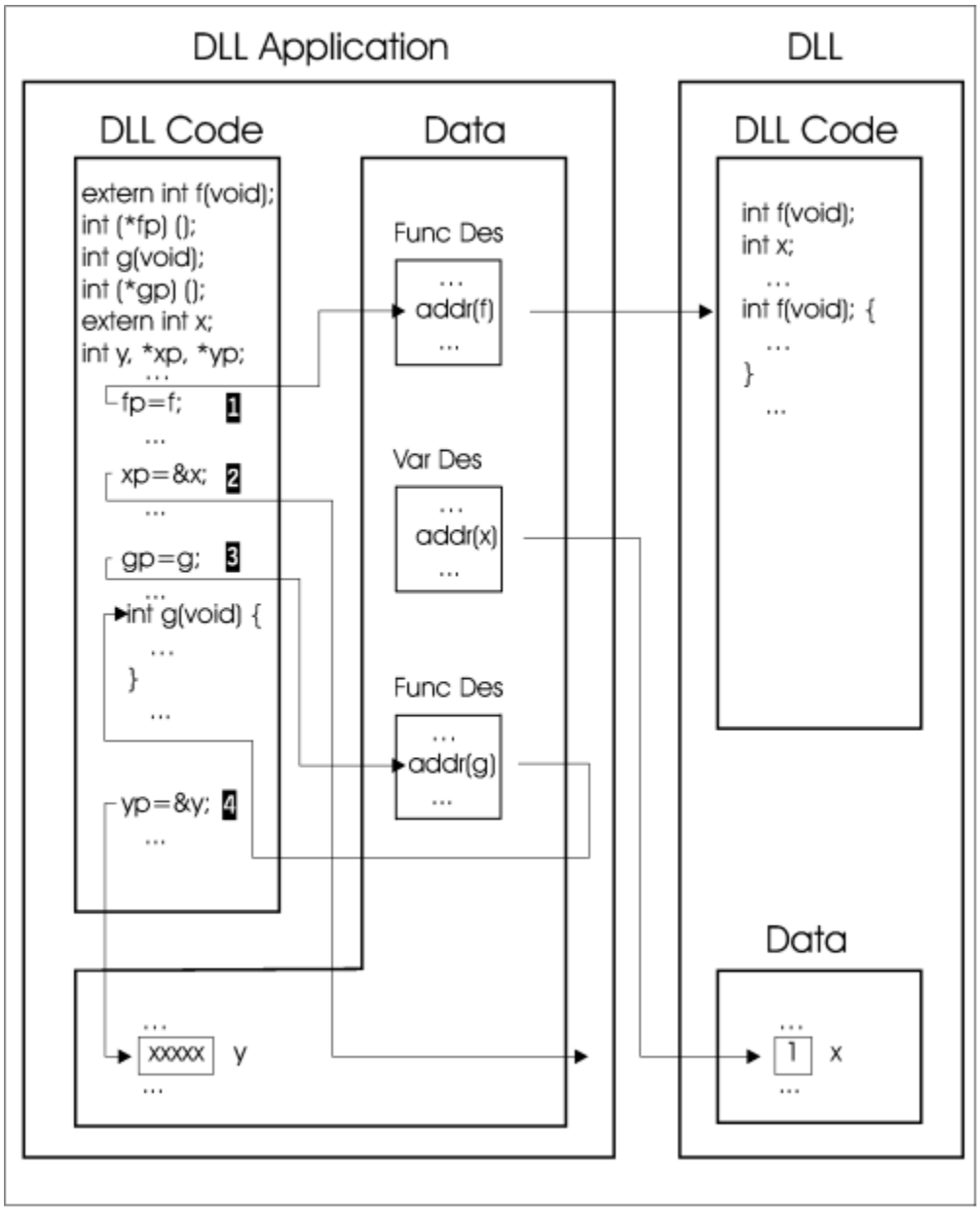


Figure 14. Pointer Assignment in DLL Code

In Figure 15 on page 100, 1 causes a link error because the assignment to `fp` is undefined. 2 causes a prelink error because the assignment to xp is undefined. 3 assigns gp to the address of the non-imported function, g. 4 assigns the address of the non-imported variable y to yp.

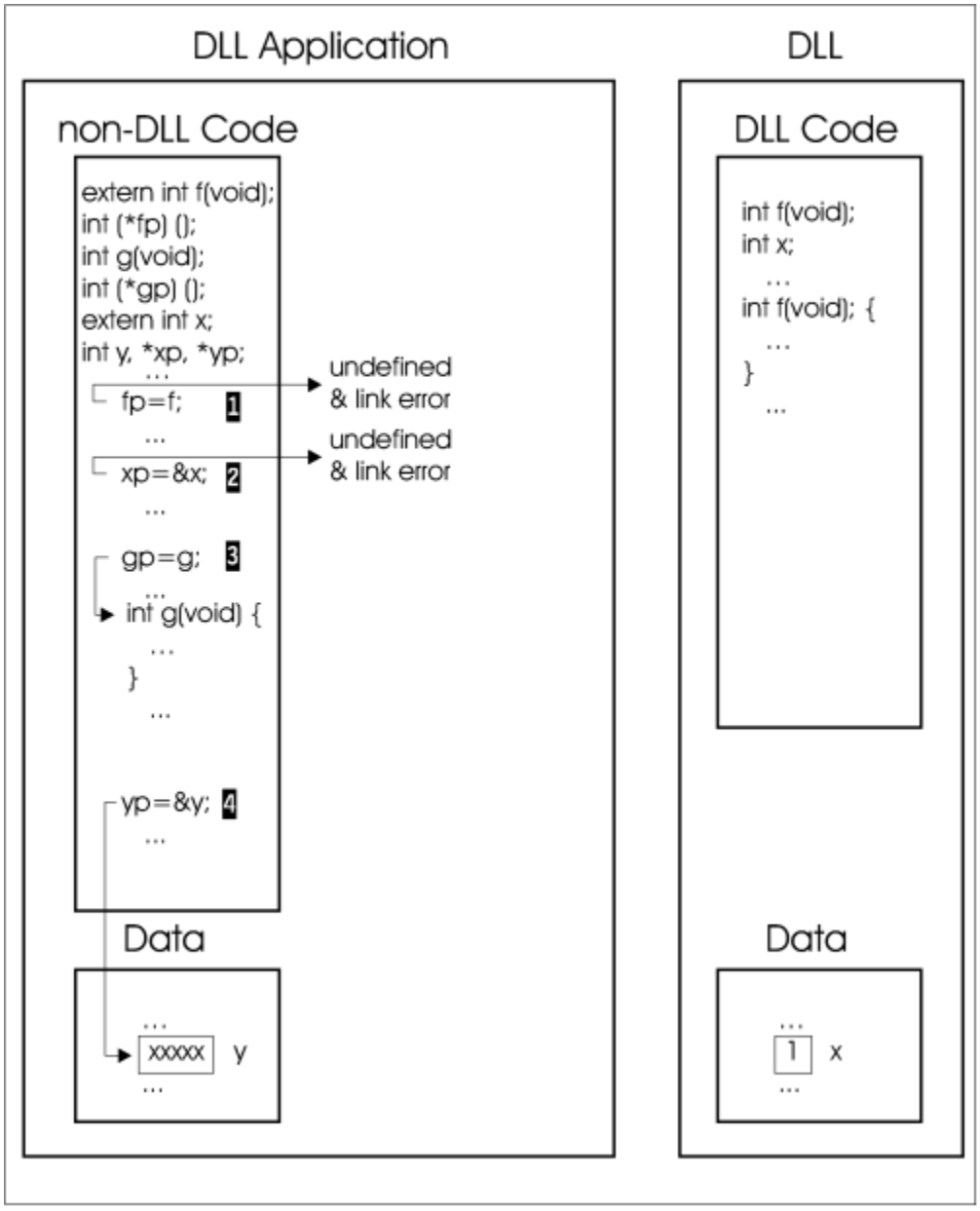


Figure 15. Pointer Assignment in Non-DLL Code

DLL Function Pointer Call in Non-DLL Code

The C/C++ compiler supports the DLL function pointer call in non-DLL code. You can, therefore, create a DLL to support both DLL and non-DLL applications. To make those calls possible, glue code is included at the beginning of a function descriptor to allow branching to a function descriptor.

A function pointer in non-DLL code points to the function entry, and a function pointer call branches to the function address. However, when a DLL function pointer is passed from DLL code to non-DLL code, the pointer points to a function descriptor. A call made through this pointer in non-DLL code results in branching to the descriptor.

The C/C++ compiler executes a DLL function pointer call in non-DLL code by branching to the descriptor and executing the glue code that invokes the actual function.

The example below and in Figure 19 on page 103 show a DLL function pointer call in non-DLL code, where a simplified `qsort()` routine is used as an example of non-DLL code. Note that the `qsort()` routine compiled as non-DLL code can be called from both a DLL application and a non-DLL application.

C Examples

Figure 16 on page 101 shows an example of C non-DLL code in a DLL.

```
int qqsort(int* z,int num,int (*comp)(int e1,int e2))
{
  int i,j,temp,rc;

  for(i=0;i<num-1;i++)
  {
   for(j=1;j<num-i;j++)
   {
    rc=(*comp)(z[j-1],z[j]);
    if(rc>0)
    {
     temp=z j ;
     z j =z j-1 ;
     z j-1 =temp;
    }
   }
  }
  return(0);
}

int comp(int e1,int e2)
{
   if(e1==e2){
     return(0);
   }
   else if(e1<e2){
     return(-1);
   }
   else{
     return(1);
   }
}
```

Figure 16. C Non-DLL Code in a DLL

Figure 17 on page 101 shows an example of C DLL code in a DLL application.

```
                  int (*comp)(int e1, int e2));
int comp(int e1, int e2)
{
  if (e1 == e2)
    return(0);
  else (e1 < e2)
    return(-1);
  else
    return(1);
}
int (*fp)(int e1, int e2);
main()
{
  int a[2] = { 2, 1 };
  fp = comp;       /* assign address of function descriptor */
  qsort(a, 2, fp); /* call qsort */
}
```

Figure 17. C DLL Code in a DLL Application

Figure 18 on page 102 shows an example of C non-DLL code in a non-DLL application.

```
                  int (*comp)(int e1, int e2));
int comp(int e1, int e2)
{
  if (e1 == e2)
    return(0);
  else if (e1 < e2)
    return(-1);
  else
    return(1);
}
int (*fp)(int e1, int e2);
main()
{
  int a[2&] = { 2, 1 };
  fp = comp;       /* assign function address */
  qsort(a, 2, fp); /* call qsort */
}
```

Figure 18. C Non-DLL Code in a Non-DLL Application

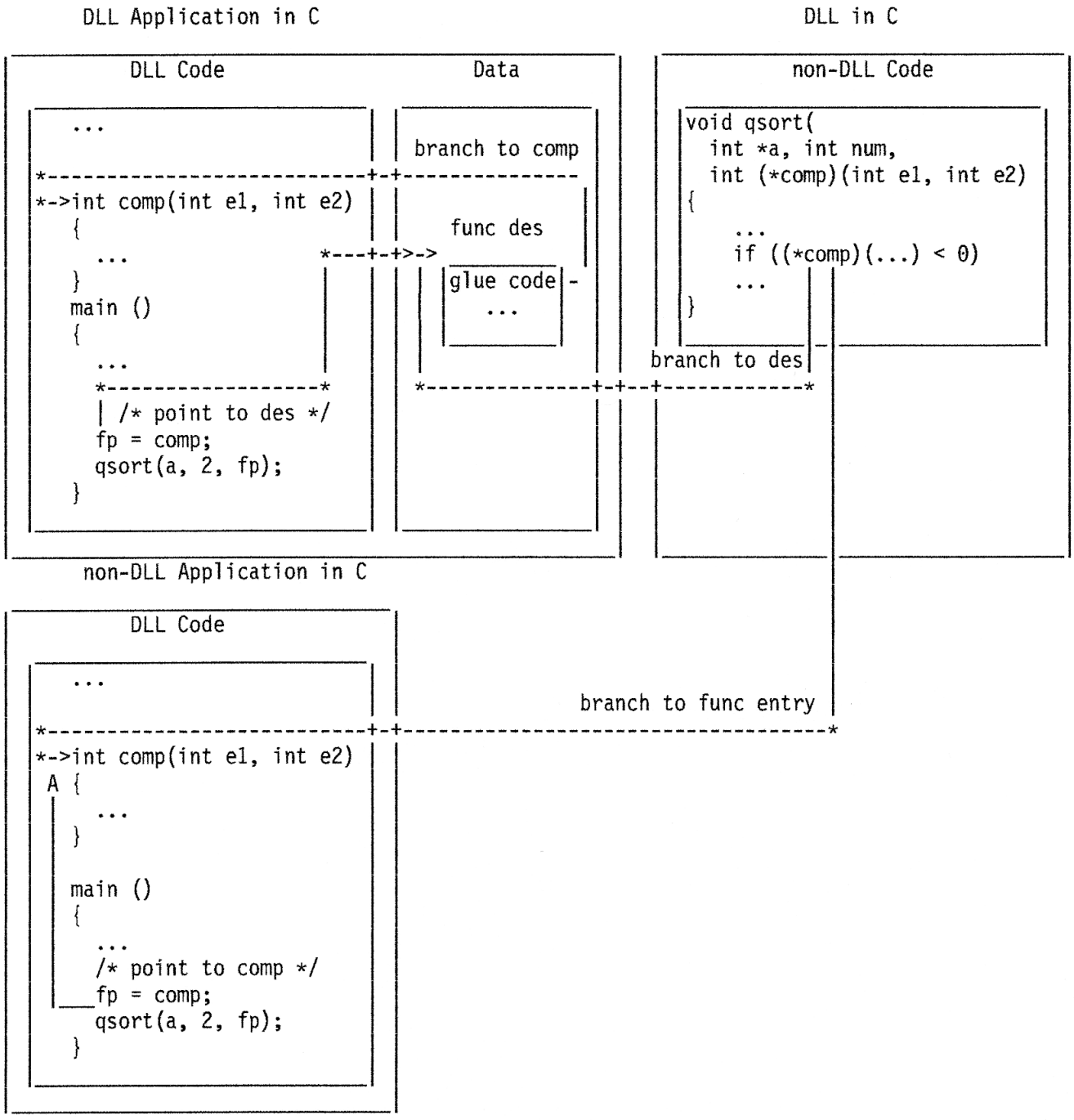


Figure 19. DLL Function Pointer Call in Non-DLL Code

Non-DLL Function Pointer Call in DLL Code

In DLL code, a function pointer is assumed to point to a function descriptor. A function pointer call is made by first obtaining the function address through de-referencing the pointer, and then branching to the function entry. When a non-DLL function pointer is passed to DLL code, it points directly to the function entry. An attempt to de-reference through such a pointer produces an undefined function address. The subsequent branching to the undefined address may result in an exception. Below are examples of passing a non-DLL function pointer to DLL code through an external variable. Its behavior is undefined:

C Examples

- C DLL code:

```
void (*fp)(void);
void main (void) {
  goo();
  (*fp)();    /* Expect a descriptor, but get a function address, */
              /* so it de-references to an undefined address and  */
              /* call fails                                       */
}
```

- C Non-DLL code:

```
extern void (*fp)(void);
void hello(void) {
  printf("hello\n");
}
void goo(void) {
  fp = hello; /* assign address of hello, to fp    */
}
```

Function Pointer Comparison in Non-DLL Code

In non-DLL code, the results of the following function pointer comparisons are undefined:

1. Comparing a DLL function pointer to a non-DLL function pointer, or, in other words, comparing the address of a function to the address of a function descriptor. In the following examples, both the DLL function pointer and the non-DLL function pointer point to the same function, but the pointers themselves compare unequal (refer to Figure 20 on page 104).

   - C DLL code:

   ```
   extern int foo(int (*fp1)(const char *, ...));
   main ()
   {
     int (*fp)(const char *, ...);
     fp = printf;  /* assign address of a descriptor that   */
                   /* points to printf.                     */
     if (foo(fp))
       printf("Test result is undefined\n");
   }
   ```

   - C Non-DLL code:

   ```
   {
     int (*fp2)(const char *, ...);
     fp2 = printf;   /* assign the address of printf.        */
     if (fp1 == fp2) /* comparing address of descriptor to   */
                     /* address of printf results in unequal.*/
       return(0);
     else
       return(1);
   }
   ```

   In the preceding example, DLL code and non-DLL code can reside either in the same load module or in different load modules.

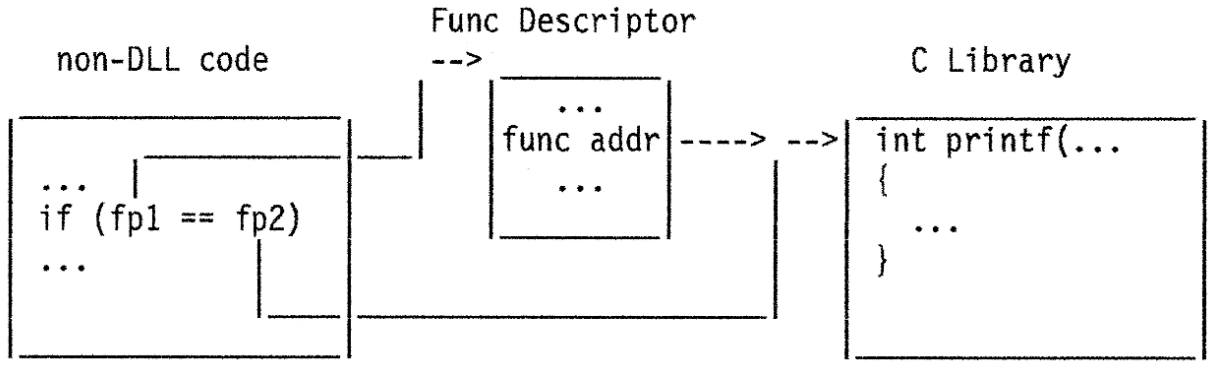


Figure 20. Comparison of Function Pointers in Non-DLL Code

2. Comparing a DLL function pointer to another DLL function pointer. Here, you are comparing addresses of function descriptors. In the following examples, both DLL function pointers point to the same function, but they compare unequal (refer to Figure 21 on page 105).

   - C DLL1 code:

```
extern int goo(int (*fp1)(const char *, ...));
main ()
{
  int (*fp)(const char *, ...);
  fp = printf;  /* assign address of a descriptor that   */
                /* points to printf.                     */
  if (goo(fp))
    printf("Test result is undefined\n");
}
```

   - C DLL2 code:

```
extern int foo(int (*fp1)(const char *, ...),
               int (*fp2)(const char *, ...));
int goo(int (*fp1)(const char *, ...))
{
  int (*fp2)(const char *, ...);
  fp2 = printf; /* assign address of a different         */
                /* descriptor that points to printf.     */
  return (foo(fp1, fp2));
}
```

   - C Non-DLL code:

```
        int (*fp2)(const char *, ...))
{
  if (fp1 == fp2) /* comparing the addresses of two       */
                  /* descriptors results in unequal.      */
    return(0);
  else
    return(1);
}
```

   Here, DLL1 code and DLL2 code reside in different load modules. The non-DLL code can reside in the same load module with DLL1 or DLL2 or in a different load module.

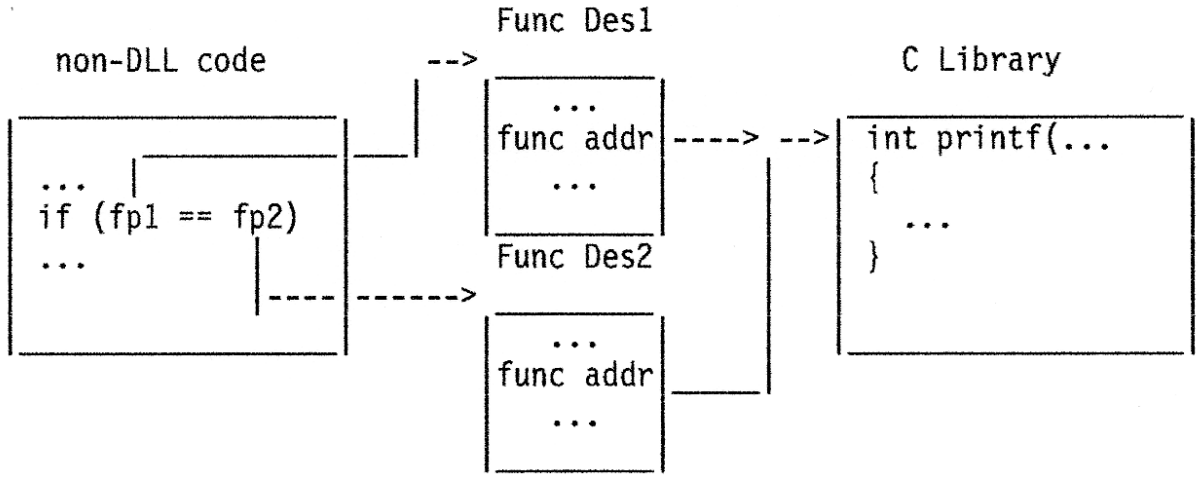


Figure 21. Comparison of Two DLL Function Pointers in Non-DLL Code

3. Comparing a DLL function pointer to a constant function address other than NULL. Here, you are comparing the constant function address to an address of a function descriptor.

   Note: Comparing a DLL function pointer to NULL is well defined, because when a pointer variable is initialized to NULL in DLL code, it has a value zero.

Function Pointer Comparison in DLL Code

In DLL code, a function pointer is checked for NULL before it is compared. For a non-NULL pointer, the pointer is further de-referenced to obtain the function address that is used for the comparison. For an uninitialized function pointer that actually has a non-zero value, the de-reference can cause an exception to occur if the storage that the uninitialized pointer points to is read-protected.

Usually, comparing uninitialized function pointers in DLL code results in undefined behavior. You must initialize a function pointer to NULL or the function address (from source view). Two examples follow.

1. Undefined comparison in DLL code:

```
int (*fp2)(const char *, ...)  /* Initialize to point to the      */
                    = printf; /* descriptor for printf           */
int goo(void);
int (*fp2)(void) = goo;
int goo(void) {
  int (*fp1)(void);
  if (fp1 == fp2)
    return (0);
  else
    return (1);
}
 #include <stdio.h>

  void check_fp(void (*fp)()) {
      /* exception likely when -1 is de-referenced below */
      if (fp == (void (*)())-1)
        printf("Found terminator\n");
      else
        fp();
  }
  void dummy() {
    printf("In function\n");
  }

  main() {
    void (*fa[2])();
    int i;

    fa[0] = dummy;
    fa[1] = (void (*)())-1;

    for(i=0;i<2;i++)
      check_fp(fa[i]);
  }
```

Figure 22 on page 106 shows that, when `fp1` points to a read-protected memory block, an exception occurs.

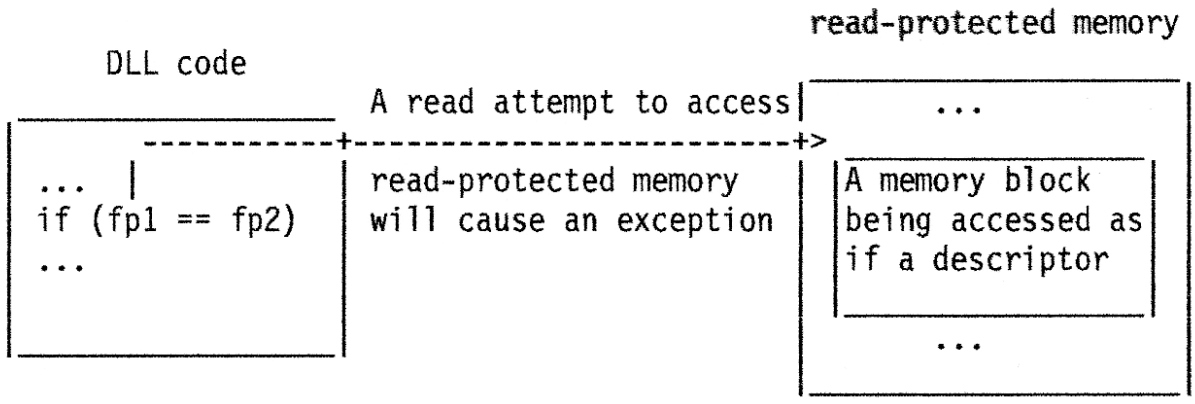


Figure 22. Comparison of Function Pointers in DLL Code

2. Valid comparisons in DLL code:

```
int (*fp1)(const char *, ...); /* An extern variable is implicitly*/
                               /* initialized to zero             */
                               /* if it has not been explicitly   */
                               /* initialized in source.          */
```

```
int (*fp2)(const char *, ...)  /* Initialize to point to the      */
                    = printf; /* descriptor for printf           */
int foo(void) {
  if (fp1 != fp2 )
    return (0);
  else
    return (1);
}
```

Explicitly Calling a DLL

DLLs can be explicitly controlled by the application code at the source level. Explicit source-level calls to one or more execution-time services connect the reference and the definition.

The DLL application writer can explicitly call the following execution-time services:

- `dllload()`
- `dllqueryfn()`
- `dllqueryvar()`
- `dllfree()`

Apart from these calls, there is no implicit or automatic connection between the application and the DLL.

The following guidelines apply to explicitly calling a DLL in your application:

- Determine the names of the exported functions and variables that you want to use. You can get this information from the DLL provider's documentation or by looking at the definition side-deck file that came with the DLL.
- Include the DLL header file, `dll.h`, in your application.
- Compile your source as usual.
- Prelink your code as usual.

  Note: Do not prelink with the definition side-deck if you are calling the DLL explicitly with the callable services.
- Link your application with the same AMODE value as the DLL.

Figure 23 on page 108 is an example of an application that explicitly uses a DLL:

```
#include <stdio.h>
#include <string.h>

#ifdef __cplusplus
  extern "C" {
#endif

  typedef int (DLL_FN)(void);

#ifdef __cplusplus
  }
#endif

#define FUNCTION        "FUNCTION"
#define VARIABLE        "VARIABLE"

static void Syntax(const char* progName) {
  fprintf(stderr, "Syntax: %s <DLL-name> <type> <identifier>\n"
                  "  where\n"
                  "  <DLL-name> is the DLL to load,\n"
                  "  <type> can be one of FUNCTION or VARIABLE\n"
                  "  and <identifier> is the function or variable\n"
                  "  to reference\n", progName);
  return;
}int value;
  int* varPtr;
  char* dll;
  char* type;
  char* id;
  dllhandle* dllHandle;

  if (argc != 4) {
    Syntax(argv[0]);
    return(4);
  }

  dll  = argv[1];
  type = argv[2];
  id   = argv[3];
```

Figure 23. Explicit Use of a DLL in an Application Part 1 of 2

```
  dllHandle = dllload(dll);
  if (dllHandle == NULL) {
    perror("DLL-Load");
    fprintf(stderr, "Load of DLL %s failed\n", dll);
    return(8);
  }   if (strcmp(type, VARIABLE)) {
      fprintf(stderr,
        "Type specified was not " FUNCTION " or " VARIABLE "\n");
      Syntax(argv[0]);
      return(8);
    }
    /*
     * variable request, so get address of variable
     */
    varPtr = (int*)(dllqueryvar(dllHandle, id));
    if (varPtr == NULL) {
      perror("DLL-Query-Var");
      fprintf(stderr, "Variable %s not exported from %s\n", id, dll);
      return(8);
    }
    value = *varPtr;
    printf("Variable %s has a value of %d\n", id, value);
  }
  else {
    /*
     * function request, so get function descriptor and call it
     */
    DLL_FN* fn = (DLL_FN*) (dllqueryfn(dllHandle, id));
    if (fn == NULL) {
      perror("DLL-Query-Fn");
      fprintf(stderr, "Function %s() not exported from %s\n", id, dll);
      return(8);
    }
    value = fn();
    printf("Result of call to %s() is %d\n", id, value);
  }
  dllfree(dllHandle);

  return(0);
}
```

Figure 24. Explicit Use of a DLL in an Application Part 2 of 2

Part 3. Using CMS Services

This part discusses the programming services available in CMS. You will use these services when you develop your applications.These services are included in the following chapters:

Chapter 10. Handling Input and Output

This chapter describes the I/O operations your application can perform. These I/O operations include:

- File I/O
- Directory I/O
- Console and Terminal I/O
- Program Stack I/O
- Unit Record I/O
- Tape I/O
- General Tape I/O Services

File I/O

Your application can manipulate CMS files using any of the following methods:

- Use record file system routines to manage SFS and minidisk files. See "File I/O" on page 149 for more information on these routines.
- Use high-level language statements for opening, reading, writing, and closing files. When using a high-level language such as FORTRAN, COBOL, or PL/I, operating on files stored in SFS is the same as operating on files stored on a minidisk.
- Use these CMS FS macros to manage CMS files: FSCB, FSCBD, FSOPEN, FSREAD, FSWRITE, FSPOINT, FSCLOSE, FSERASE, and FSSTATE.

  These macros can manipulate files in SFS or on a minidisk. Applications written in a high-level language must call assembler subroutines that use these macros. See the *z/VM: CMS Application Development Guide for Assembler* for information on using the FS macros.
- Use the EXECIO command or the PIPE command command to read and write information.

  The PIPE command can be used from a REXX or EXEC 2 exec. The following stage commands, which are operands on the PIPE command, read and write records:

  <
  reads the contents of a CMS file.

  >
  writes to (replaces or creates) a CMS file.

  >>
  writes to (appends to or creates) a CMS file.

  FILEFAST
  reads from or writes to a CMS file.

  FILEBACK
  reads from a CMS file backwards.

  FILERAND
  reads specific records from a CMS file.

  FILESLOW
  reads from or writes to a CMS file beginning at a specified record.

  See the *z/VM: CMS Pipelines User's Guide and Reference* for details on these stage commands.

  The EXECIO command is normally issued from a REXX or EXEC 2 exec. See the *z/VM: CMS Commands and Utilities Reference* for details on the EXECIO command.
- Use OS and DOS simulated macros to manipulate files in SFS or on a minidisk. These OS and DOS simulated macros use the same CMS I/O routines that the FS macros do. See the *z/VM: CMS Application*

Development Guide for Assembler for information on using OS and DOS macros simulated macros to access files.

Your application can manipulate BFS files using the following methods:

- Use OpenExtensions callable services. For more information about these routines, see the *z/VM: OpenExtensions Callable Services Reference*.
- Use the CMS Pipelines PIPE command. The PIPE command runs a series of stage commands called a pipeline. CMS Pipelines stages <, > and >> for reading and writing disk files transparently use the proper low-level device driver when the file is a BFS file. The low-level device driver will transparently transform between BFS data stream and records in the pipeline.
- Use CMS record file system CSL routines. This support is primarily for administration and system-managed storage purposes. See Chapter 13, "Manipulating BFS Files and Directories Using CMS Record File System CSL Routines," on page 193.

Directory I/O

Your application can use CSL routines to perform the following tasks on SFS and minidisk directories:

- Determine if a directory exists
- Open a directory to read it
- Read directory records
- Create or erase a directory
- Close a directory
- Commit your changes.

See "Directory I/O" on page 165 for information on these routines.

To perform I/O on BFS directories, use the OpenExtensions callable services. See the *z/VM: OpenExtensions Callable Services Reference*. You can also use CMS record file system routines for administration and system-managed storage purposes. See Chapter 13, "Manipulating BFS Files and Directories Using CMS Record File System CSL Routines," on page 193.

Your application can use CMS Pipelines stage commands to perform the following tasks on BFS directories:

BFSDIRECTORY
reads from an existing BFS directory file and writes one record for each directory entry.

BFSQUERY
obtains information from OpenExtensions about the current working BFS directory

BFSSTATE
writes records containing status information about byte stream files

BFSXECUTE
reads a record containing a request and sends that request to OpenExtensions services.

Here is a partial list of request tasks:

- Change a directory
- Create a new directory entry
- Create, rename, or remove a directory

Console and Terminal I/O

Your application can use the following methods to create an interactive application:

- Use the XMITMSG command. XMITMSG accesses a message from a CMS message repository file or your own message repository file and displays the message. For information on using the XMITMSG command, see "Creating and Using Message Repositories" on page 405.

- Use the Interactive System Productivity Facility (ISPF) and the Display Management System for CMS (DMS/CMS)

  ISPF and DMS/CMS provide services to make your application interactive. For details on ISPF and DMS/CMS, see Chapter 24, “Creating an Interactive Program,” on page 369.

- Use the CONSOLE, LINERD, and LINEWRT macros.

  CONSOLE
  provides CMS fullscreen console services, which includes 3270 I/O operations. See the *z/VM: CMS Application Development Guide for Assembler* for details on using the CONSOLE macro to access CMS fullscreen console services.

  LINERD and LINEWRT
  LINERD reads a line of input from the terminal. LINEWRT displays a line of output at a terminal. See the *z/VM: CMS Application Development Guide for Assembler* for details on using the LINERD and LINEWRT macros.

  Applications written in a high-level language must call assembler subroutines that use these macros.

- Use the CMS Pipelines stage commands FULLSCREEN, BUILDSCR, CONSOLE, FULLSCRQ, FULLSCRS, APLENCODE, APLDECODE, 3270ENC, or 3270BFRA, which are operands on the PIPE command.

  FULLSCREEN
  writes 3270 data streams to the virtual console in fullscreen mode or to a 3270 device.

  BUILDSCR
  builds 3270 data streams.

  CONSOLE
  reads from or writes to the terminal in line mode.

  FULLSCRQ
  provides information about the terminal.

  FULLSCRS
  processes the output from FULLSCRQ to make it easier to use.

  APLENCODE
  reads its primary input stream records and translates a single character into a graphic escape character sequence that can be displayed on a 3270 terminal capable of displaying APL/TEXT characters.

  APLDECODE
  reads its primary input stream records that contain data from a 3270 terminal capable of displaying APL/TEXT characters and translates a graphic escape character sequence into a single character.

  3270ENC
  prepares a 64 character translate table used to convert binary values in the range B'000000' through B'111111' (64 values) to displayable 1-byte graphic characters for placement in a 3270 data stream.

  3270BFRA
  converts a 2-byte unsigned integer (the format of the 14- and 16-bit buffer addresses) to a 12-bit buffer address, or vice versa.

  See the *z/VM: CMS Pipelines User's Guide and Reference* for details on these stage commands.

OS/MVS™ Simulation provides WTO and WTOR macros to support terminal I/O. See the *z/VM: CMS Application Development Guide for Assembler* for more information.

Program Stack I/O

Your application can use these methods to manipulate the CMS program stack.

- CMS commands

  SENTRIES
  Returns the number of line in the program stack.

MAKEBUF
Adds a buffer to the program stack.

DROPBUF
Removes buffers from the top of the program stack.

DESBUF
Deletes all lines in the program stack, lines in the terminal input and output buffers, and pending terminal input.

- CMS macros

CMSSTACK
Writes a line to the program stack.

LINERD
Reads a line from the program stack or the terminal input buffer.

- CSL routines

StackBufferCreate
Adds a buffer to the program stack.

StackBufferDelete
Removes buffers from the top of the program stack.

StackQuery
Queries the number of lines and number of buffers in the program stack.

StackRead
Reads a line from the program stack.

StackWrite
Writes a line to the program stack.

- CMS Pipelines STACK stage command.

Commands are documented in the *z/VM: CMS Commands and Utilities Reference*, macros in the *z/VM: CMS Macros and Functions Reference*, CSL routines in the *z/VM: CMS Callable Services Reference* , and CMS Pipelines in the *z/VM: CMS Pipelines User's Guide and Reference*.

Unit Record I/O

Your application can use the following methods to write information to a virtual printer, write information to a virtual punch, or read information from a virtual reader:

- Use the following macros:

PRINTL
writes information to a virtual printer.

PUNCHC
writes information to a virtual punch.

RCARD
reads information from a virtual reader.

Applications written in a high-level language must call assembler subroutines that use these macros. See the *z/VM: CMS Macros and Functions Reference* for more information on these macros.

- Use the following stage commands, which are operands on the PIPE command, in a REXX or EXEC 2 exec:

PRINTMC
writes information to a virtual printer.

PUNCH
writes information to a virtual punch.

URO
writes information to a virtual printer or virtual punch.

READER
reads information from a virtual reader.

See the *z/VM: CMS Pipelines User's Guide and Reference* for details on these stage commands.

Tape I/O

Your application can use the following methods to use tapes:

- Use the following macros:

  WRTAPE
  writes a block on a tape drive.

  RDTAPE
  reads a block from a tape drive.

  TAPECTL
  positions the tape according to the specified function code.

  TAPESL
  processes IBM standard HDR1 and EOF1 labels.

  Applications written in a high-level language must call assembler subroutines that use these macros. See the *z/VM: CMS Macros and Functions Reference* for information on the WRTAPE, RDTAPE, TAPECTL, and TAPESL macros.

- Use the following commands:

  TAPE DUMP
  dumps (writes) CMS-formatted files from a disk or directory to tape.

  TAPE LOAD
  loads (reads) previously dumped files from tape to a disk or directory,

  TAPE SCAN
  positions the tape at a specified point. Scanning stops upon encountering the file.

  TAPE SKIP
  positions the tape at a specified point. Skipping stops upon encountering the file.

  VMFPLC2
  loads files from product tape and service tapes, dumps CMS files to tape, and loads previously dumped files from tape.

  See the *z/VM: CMS Commands and Utilities Reference* for details on the TAPE and VMFPLC2 commands.

- Use the TAPE stage command, which is an operand on the PIPE command, to read from or write to a tape.

  See the *z/VM: CMS Pipelines User's Guide and Reference* for details on this stage command.

General Tape I/O Services

CMS OS/MVS Simulation provides several macros and commands to support tape I/O. See the *z/VM: CMS Application Development Guide for Assembler* and the *z/VM: CMS Commands and Utilities Reference* for more information.

In addition, DFSMS/VM provides a group of Removable Media Services (RMS) Tape Library Dataserver interface CSL routines (in FSMPPSI CSLLIB). These routines allow applications running under CMS OS simulation to issue requests for Tape Library Dataserver functions such as mounting or demounting a tape, querying library information, setting or resetting the device category, and setting the volume category. For information on using Tape Library Dataservers under OS simulation, see *z/VM: CMS Application Development Guide for Assembler*. For descriptions of the DFSMS/VM RMS interface routines, see *z/VM: DFSMS/VM Removable Media Services*.

Chapter 11. Understanding the CMS File System

This chapter discusses:

- The CMS record file system architectures
- CMS support for the Byte File System (BFS)
- The attributes of CMS record files and BFS files in CMS
- The interfaces you can use to manipulate CMS record files and BFS files.

Reviewing these concepts should help you design your applications to make the best use of CMS files and BFS files.

File System Architectures Supported by CMS

CMS supports two record file system architectures—Enhanced Disk Format (EDF) and Shared File System (SFS). CMS also supports the OpenExtensions Byte File System (BFS).

Note: CMS clients can access all three CMS file systems using the Network File System (NFS) protocol. For more information, see *z/VM: TCP/IP User's Guide*.

Enhanced Disk Format (EDF) Architecture

A ***virtual disk***, or ***minidisk***, is a place where you can collect files. Files are what you 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, 1KB, 2KB, or 4KB 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 this 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 upon its intended use. A 4KB block size optimizes the I/O if the disk is to contain large files with no missing records (dense). A block size of 1KB is more appropriate when creating many small files or files with missing records (sparse). For example, PL/I regional files are sparse and they may allocate more space on a 4KB disk than on a 1KB disk; therefore, the smaller block size is preferable.

The block size of the disk can affect the amount of storage required for I/O buffers. When caching is not in effect, a larger block size will always result in a greater amount of storage required for the I/O buffers. When caching is in effect, the storage requirements are not determined by the block size of the disk.

Shared File System (SFS) Architecture

SFS files are represented in storage using a pointer block structure that is similar to the EDF structure. Unlike EDF files, however, SFS files can span minidisks. That is, the blocks that make up an SFS file can reside on more than one minidisk within a ***file pool***. A file pool is a collection of minidisks that is owned and manipulated by a file pool server machine. In SFS file architecture, the pointer blocks do not refer to physical disk locations. Instead, they are relative (or logical) numbers. The file pool server uses control data, which the server maintains, to find the appropriate minidisk blocks based on these logical numbers. All SFS files are formatted with 4KB blocks. See Chapter 12, “Manipulating SFS and Minidisk Files and Directories,” on page 129 for more information on SFS.

OpenExtensions Byte File System (BFS)

OpenExtensions is the VM implementation of IEEE POSIX standards for system interfaces and threads. Included in OpenExtensions is a POSIX-compliant file system called the Byte File System (BFS). BFS is a companion to SFS that provides a byte-stream view of files. That is, a BFS file consists of an ordered sequence of bytes rather than records. 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 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 placed in virtual storage that is available to 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. You can retrieve the attributes of files using the Callable Services Library (CSL) routines. See the *z/VM: CMS Callable Services Reference* for information on retrieving file attributes. 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** for files stored in file pools:

- File space type (SFS or BFS)
- Recoverability
- Overwrite
- Date of last reference
- Creation date and time
- Date and time of last change.

These extended file attributes are not maintained in the FSTs. Your programs can retrieve them by using CSL routines.

BFS files have two sets of file attributes associated with them. The first set consists of the path name, size in bytes, permission bits, and other attributes associated with the file in accordance with the IEEE POSIX 1003.1 standard. Because BFS files are stored in a CMS file pool, 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 presented to the record file system CSL routines that manipulate them as fixed-length record-format files with a logical record length of 1.

File Name, File Type, and 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 complete discussion of valid file names, file types, and file modes.

Every CMS file, regardless of whether it resides in an SFS directory 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.[1] For the SFS environment, file mode number 0 means the same as file mode number 1.

File Mode Number 1
File mode number 1 is used for reading and writing files. It is the default file mode number.

File Mode Number 2
File mode number 2 is the same as file mode 1.

File Mode Number 3
File mode number 3 means that 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 SFS directories.

File Mode Number 4
File mode number 4 means files are in OS simulated data set format. These files are created by OS macros in programs running in CMS.

File Mode Number 5
File mode number 5 is the same as file mode number 1.

File Mode Number 6
For EDF files, file mode number 6 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.

Attention: When modifying an existing file mode number 6 file, it is possible to corrupt the file, or even the entire minidisk on which it resides. This corruption 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 more information on the EDF update-in-place attribute and EDF data integrity, see the *z/VM: CMS Application Development Guide for Assembler*.

For SFS files, file mode number 6 is the same as file mode number 1. You can also give SFS files the update-in-place (or INPLACE) attribute. This means that the existing records of a file are written back to their previous location in the file pool, rather than in a new slot. See "Overwrite" on page 123 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 BFS 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 the DMSOPDIR routine with an intent of FILEEXT or by using the OPENVM LISTFILE command with the NAMES option.

The CMS file mode number of a BFS file, when it applies, is always 1.

[1] Normally, if someone links to your disk in read-only mode and requests a list of all the files on your disk, the files with file mode number 0 are not accessible unless the ACCESSM0 ON command has been specified. However, the DDR command could be used to copy a whole minidisk from one disk to another. In this situation, all files with file mode number 0 are also copied.

Record Formats

From the user's point of view, a non-empty CMS file consists of one to 2,147,483,647 (2^{31}-1) records, each of which consists of 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 is usually 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. 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 can not be changed by any subsequent write to the file. The length of each record in a V-format file is recorded by a 2-byte length prefix stored immediately before the record itself. A record must not be null, for example, 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.

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 that of the preceding record. Thus, the number of records in a file is the greatest number of any record written.

An SFS file may be empty (contains zero records). A minidisk file always has at least one record.

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.

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 are used to 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 and on the size of the file.

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).

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. A privileged user entering the CP SET TIMEZONE command alters the time stamps recorded on new or updated files.

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 recoverable or nonrecoverable. The recoverability attribute is only valid for files that reside in SFS file pools.

When a file is recoverable, uncommitted changes are backed out as the result of an application initiated rollback. Files are generally recoverable unless otherwise specified. See “Using the Recoverability and Overwrite Attributes” on page 142 for information on changing attributes.

Nonrecoverable files are not rolled back in the event of an application initiated rollback. As many updates as possible are committed.

BFS files are recoverable when using CMS record file interfaces.

Overwrite

The overwrite attribute specifies whether updates to a file are made in place. The overwrite attribute is valid only for files that reside in CMS file pools.

Most files are not updated in place (NOTINPLACE files). That is, when you change data in a file block, the original block in the file pool is not changed. Instead, the file pool server allocates a new block to contain the changed information. This process is known as ***shadowing***.

By using shadowing, the file pool server is able to provide different consistent views of a file to the writer and any readers. When readers close the unmodified version of the file, the old blocks are made available for other use and the changed blocks become the new version of the file. SFS always shadows file updates unless you specify otherwise.

With INPLACE files, updates are made in place where possible. The SFS file pool server does not use shadowing. When data in a file block is changed, the original block in the file pool is changed. Once the file pool server writes the block to the file pool, it is immediately available for readers to see. The file pool server, in this case, does not provide readers with a consistent view of the data from open to close. A reader of an INPLACE file can read the same record twice while the file is open and see different data. (This is true regardless of whether the file resides in a file control or in a directory control directory.)

Although blocks are made available to readers as they are written to a file pool, blocks you change may not be immediately transmitted to the file pool server. By default, CMS maintains a buffer of file blocks in your virtual storage. Blocks that you change may be temporarily held in the buffer. When the buffer is full or when you close the file, changed blocks are transmitted to the file pool server. The file pool server then replaces the file pool blocks and the changes are made available to others.

Because buffers are also used for readers, readers may not immediately see the changed file pool blocks. CMS does not request a block from the file pool server if the block already exists in the buffer. You can circumvent the use of buffers on inplace files, using the FORCE parameter when writing records (DMSWRITE routine) and the REFRESH parameter when reading records (DMSREAD routine). The FORCE parameter causes a changed block to be immediately transmitted to the file pool server, while REFRESH causes CMS to get the latest block from the file pool server. Naturally, this does not perform as well as reading from and writing to a buffer, so you should use FORCE/REFRESH only when necessary.

To avoid corrupting files, file pool servers always shadow file pointer blocks. One important result of this is that readers of INPLACE files cannot see file extensions unless they close and reopen the file. If the file resides in an accessed directory control directory, readers must reaccess the directory to see the extensions. A ***file extension*** is any new block or record that is added to the file. An added record includes not only records that are appended to the file, but may include those that have written to previously-unwritten records in the middle of a file.

Example: Suppose you create a file with a fixed-length record format by writing one record to it. That record happens to be record 30000. Now someone opens the file to read it. That person can read record 30000 and see the data. Reading any record from 1 to 29999 would cause binary zeros to be returned. If you open the file and write record 5000, the reader who already had the file open would not see what you have written. Even though the file is an INPLACE file, the reader would still receive binary zeros. The addition of the record in the middle of the file is considered a file extension, so the reader does not see the change.

Because the file extensions are shadowed, you may want to avoid extending INPLACE files for certain applications. Instead, preformat the file for the application by writing blank records. Do not write records of binary zeros when preformatting files. CMS interprets these as null records and does not write them to data blocks (that is, they do not occupy physical file space).

BFS files are always NOTINPLACE.

Date of Last Reference

The date of last reference attribute specifies the date on which the file was last read or updated. If the file has not been read or updated since it was created, the date of last reference is the date of file creation. CMS maintains the date of last reference only for files that reside in file pools.

The difference between the date of last reference attribute and the date attribute is that the date of last reference is updated when a file is read—the date attribute is not.

The date in the date of last reference attribute is based on Coordinated Universal Time (UTC) at the time of the reference. The date in 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 GMT 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 to retrieve the date of last reference for a file. You can also use CSL routines to inhibit the updating of the date of last reference. The date of last reference is intended for use by application programs. It is also displayed by the FILELIST and LISTFILE commands through the use of the ALLDATES option. See “Using the Date of Last Reference Attribute” on page 125 for more information.

Creation Date and Time

The creation date and creation time attributes specify the date and time when the file was created. CMS maintains the creation date and time only for files that reside in file pools.

When determined by the system, the creation date and time are based on Coordinated Universal Time (UTC). You can, however, specify a creation date and time of your own choosing when you use CSL routines to create a file. These values will be viewed in Coordinated Universal Time (UTC). Once the file is created, you cannot change the creation date and time.

For BFS files, these attributes have 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 - File Extended) to retrieve the creation date and time for a file. The creation date and time are intended for use by application programs. They are also 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 LISTFILE command through the use of the ALLDATES option.

Using the Date of Last Reference Attribute

This section describes how SFS maintains the date of last reference attribute and how your program can use it. If you are not familiar with the date of last reference attribute, read the definition in “Date of Last Reference” on page 124 before proceeding.

How SFS Maintains the Date of Last Reference

The date of last reference is for applications that automatically archive files. Such applications would archive and optionally erase files that have not been referenced since a certain date. Although other applications can use the attribute, the method SFS uses to maintain it is oriented toward archiving applications.

One result of this orientation is that the file pool server seldom updates the date of last reference immediately after the reference is made. Instead, it waits until a number of files in the file pool have been referenced. Then the file pool server updates its catalog data with the new dates for all the referenced files. By grouping the updates together, the file pool server performs better than it would if it updated the catalogs as the references occurred.

While this design optimizes the file pool server performance, there may be some delay between the time the file is referenced and the time the attribute is updated. If you retrieve the date of last reference before the catalogs are updated, you will see the old date of last reference. This delay is not important for archiving applications—in most cases, the attribute would be updated by the time the archive job is run. If the update does not occur until after the archive, the next archive would see the update.

Also for improved performance, the file pool server does not do any extra work to ensure that scheduled attribute updates are made in the event of a system failure. If server processing ends abnormally, any scheduled updates to the attributes are lost. The file pool server makes the updates frequently, however, so only a dozen updates (at most) might be lost.

If your application fails or rolls back the work unit associated with the file reference, the date of last reference is still updated. The file pool server considers a file to be referenced when it is successfully opened, regardless of whether the file is later closed or committed. In most cases, the date of last reference is the date the file was opened. If, however, the file is closed on a later date than the open, the server uses the date the file was closed. The file pool server never records the dates of commits and rollbacks.

Any number of CMS commands, macros, and CSL routines can cause a file to be opened. For maintaining the date of last reference, the file pool server does not distinguish among them. No matter what caused the file to be opened, the file pool server still considers it a reference.

For an alias, the date of last reference is not updated when either the alias or its base file is referenced. A reference to an alias updates the date of last reference of its base file without altering the date of last reference of the alias. At the time of its creation, an alias acquires the date of last reference of its base file, and this value does not change.

Note that the file pool server never back-dates the date of last reference. That is, it never changes the date of last reference to an earlier date.

How to Retrieve the Date of Last Reference

There are two ways to retrieve the date of last reference attribute for a file:

- Testing for existence of the file
- Retrieving directory information for the file.

Both ways involve the CSL routines such as DMSEXIFI and DMSGETDX. Another way to retrieve the information is with the FILELIST or LISTFILE command with the ALLDATES option.

How to Inhibit the Updating of the Date of Last Reference

There are four CSL routines that let you inhibit the updating of the date of last reference. Unless you use these routines to reference the file, there is no way to inhibit the updating of the date of last reference. The routines are DMSFILEC (Filecopy), DMSOPEN (Open), DMSOPDBK (Open Data Blocks) and DMSOPBLK (Open Blocks). Each of these routines have an OLDDATEREF parameter that, when specified, inhibits the updating of the date of last reference.

For DMSOPEN, DMSOPDBK, and DMSOPBLK, the OLDDATEREF parameter can be specified only when the file is being opened for reading. For DMSFILEC, the OLDDATEREF parameter applies only to the source file. There is no way to inhibit the updating of the attribute when a file is opened for any kind of write activity.

Application Interfaces

CMS provides three interfaces (sets of routines or macros) that allow applications to create and manipulate CMS files and BFS files:

- Record file system CSL routines work on minidisk and SFS files; they also work, with limitations, on BFS files.
- FS macros can manipulate minidisk and SFS files but not BFS files.
- OpenExtensions CSL routines work only on BFS files.

CMS also simulates OS and DOS/VSE macros that manipulate CMS files.

For additional information, see the following sources:

- For information on file I/O, see Chapter 10, “Handling Input and Output,” on page 113.

- For information on using the CMS file system, see Chapter 12, “Manipulating SFS and Minidisk Files and Directories,” on page 129.
- For a comparison of using CSL routines or FS macros, see the *z/VM: CMS Application Development Guide for Assembler*.
- For general information on the OpenExtensions Byte File System, see the *z/VM: OpenExtensions User's Guide*.
- For information on the OpenExtensions CSL routines, see the *z/VM: OpenExtensions Callable Services Reference*.
- For information on using the CMS record file system interface for BFS objects, see Chapter 13, “Manipulating BFS Files and Directories Using CMS Record File System CSL Routines,” on page 193.
- For information on using OS and DOS/VSE macros, see the *z/VM: CMS Application Development Guide for Assembler*.

Chapter 12. Manipulating SFS and Minidisk Files and Directories

The Shared File System (SFS) is a CMS file system for storing and managing files and for sharing CMS programs and data among users. SFS data is kept by a CMS file pool server virtual machine on server-owned disk space, which is shared among all owners of files in that file pool. Requests for data from a CMS user virtual machine are sent across an APPC/VM link to the server machine. Requests can originate either as commands or routine calls.

This chapter describes the following features and functions of the CMS file system:

- Considering the effects of DFSMS/VM*, a storage management facility of z/VM
- Default considerations for directory identifiers
- Reading and writing CMS files
- Reading and creating directories
- Sharing files and directories.

Along with SFS, files can also be stored on CMS minidisks. Minidisk files cannot be shared as easily as SFS files. In fact, CMS does not control multiple access to minidisk files by more than one user. The CSL routines allow applications to access both minidisk and SFS files. They also allow read access to the contents of minidisk directories. These routines also allow users to utilize the extended functions of SFS.

SFS and minidisks comprise the CMS record file system. Data may also be stored in the OpenExtensions Byte File System (BFS). BFS files also reside in CMS file pools. See Chapter 13, “Manipulating BFS Files and Directories Using CMS Record File System CSL Routines,” on page 193.

CMS Record File System Programming Interface

The programming interface for SFS and minidisk files is a call interface composed of CSL routines in VMLIB CSLLIB. The routines can be called from high-level languages such as COBOL, VS FORTRAN, PL/I, VS Pascal, and C, as well as REXX and assembler. (Assembler language macros are not provided. However, the file system macros (FS macros) can be used in certain nonsharing and sharing cases. For more information, see the *z/VM: CMS Application Development Guide for Assembler*.)

The programming interface for SFS has the following general characteristics:

- It accepts either file names, directory names, file mode letters, or ***namedefs***.
- It does not automatically commit changes at file close time. All changes must be explicitly committed or rolled back. See “Committing and Rolling Back Changes in Application Programs” on page 141 for more details.
- It provides recovery in the event of system or program abends. Applications can rollback (undo) changes for SFS files.

The programming interface for minidisks has the following general characteristics:

- It accepts file names, file types, file mode letters, or namedefs.
- Changes to minidisk files are "committed" when the last file that was opened for output (new, write or replace) is closed. See “Committing and Rolling Back Changes in Application Programs” on page 141 for more details.
- Changes to minidisk files are not associated with CMS workunits thus they may not be rolled back after they are made.

Before using the CSL routines in VMLIB, DMSCSL must be linked to the program using the LOAD/INCLUDE command or the LKED commands. When using the CSL routines in VMLIB, remember that they must be linked to your program. Additional language-specific statements may be necessary so that language

compilers can provide the proper assembler interface. Other programming notation, such as variable declarations, are also language-dependent. For an example of how to link to DMSCSL from each specific language, see the appropriate appendix.

Note: Some examples in this chapter are written in pseudo-code to show you the general sequence of operations. When you code your program, be aware of the requirements of the programming language that you are using.

In this chapter, the example calls to DMSCSL use the actual CSL routine name and the following variables:

retcode
The return code from CMS (a signed fullword). The return code is also placed in register 15. The return code can be:

0
Normal—the routine executed successfully.

4
Warning—the routine executed, but the result may or may not be as intended.

8
Error—If the commit parameter was specified on the routine call, a return code of 8, reason code of 50500 means that the routine did execute, but the commit was not performed. Otherwise a return code of 8 means that the routine did not execute.

12
Error—the routine did not execute and work within the work unit ID was rolled back.

16
Severe Error—the commit was performed, but the state of coordinated resources may not be consistent.

20
Severe Error—the work was rolled back, but the state of coordinated resources may not be consistent.

Return codes that are greater than 8 can only be received when there is an error in an operation to the Shared File System or other CRR participant. Errors that apply to a minidisk will return an error code of 8. You may also receive return codes from DMSCSL that are negative values. For more information on these return codes, see the *z/VM: CMS Callable Services Reference*.

reascode
The reason code from CMS (a signed fullword). The reason code explains the warning or error and is also placed in register 0. For a description of the reason codes, see the specific routines in the *z/VM: CMS Callable Services Reference*.

length
The fullword *length* parameter specifies the length of the preceding character parameter.

DFSMS/VM and SFS File Management

DFSMS/VM is an optionally-installed facility of z/VM that can help automate storage management tasks for SFS files. Files can be assigned attributes that tell DFSMS/VM how long to maintain the file. Files can be automatically deleted or moved to DFSMS/VM-owned storage automatically, to more efficiently use the available storage.

You may want to ask your SFS administrator if DFSMS/VM has been installed on your system and is being used to manage SFS files, because this can affect the behavior of files in an SFS file pool. (See *z/VM: DFSMS/VM Planning Guide* for an explanation of how this product is used.)

Movement of SFS Files by DFSMS/VM

Some SFS files that appear to reside in your file pool may actually have had their data moved into a storage repository managed by DFSMS/VM. (Note that such files still are considered by SFS to consume their usual amount of room in your file space.) These files are said to be in ***migrated status*** in your file

pool. (Files in directory control directories are never moved by DFSMS/VM.) You can identify files that have been placed in migrated status by using the SHARE option with FILELIST or LISTFILE. Files shown with an asterisk in the 'type' column are in DFSMS/VM migrated status.

These files behave exactly like regular SFS files, but they must be recalled (either automatically or explicitly) into your actual file pool before you can reference the data. This may cause a delay, depending on your system configuration and workload. Automatic recall is governed by the CMS SET RECALL command. If SET RECALL is ON (the default), recall happens automatically when the file data is referenced. If SET RECALL is OFF, the file is not recalled. You receive an error indicating that the file is migrated and not available. You can add the SET RECALL setting to your PROFILE EXEC. Explicit recall is performed with the DFSMS RECALL command. (See *z/VM: CMS Commands and Utilities Reference* for more information about SET RECALL and *z/VM: DFSMS/VM Storage Administration* for information about DFSMS RECALL.)

A file does not need to be recalled unless you need to access the file data itself (for example, with the XEDIT command). For example, you may create aliases on a file, and query or change a file's attributes (LISTFILE, FILELIST, FILEATTR, GRANT AUTHORITY, etc.) without recalling the file. The file also does not need to be recalled to be erased or to have its data replaced with COPYFILE (REPLACE.

Automatic File Movement and Erasure by DFSMS/VM

You should be aware that DFSMS/VM can automatically cause SFS files to be placed in migrated status (that is, move the file data into its storage repository) or erased at certain predetermined times, without advance warning, according to your installation's storage management policies. (Files in directory control directories **may** be erased.) File erasure criteria are usually related to how long the file has existed, or the length of time since it has last been referenced. The entire file may be erased, or only the data in the file. See *z/VM: DFSMS/VM Storage Administration* for more information about DFSMS/VM.

Determining the File Pool Server Level

Additional CMS file pool server function was added in newer releases of VM/ESA. The file pool server does not have to be at the same CMS release level as the virtual machine running an application which uses data residing in that file pool. (Different levels of CMS can be used in the same system, or the application and the file pool server may be in different systems which are using different levels of CMS.) If the application is using a function that was not in the initial release of the file pool server, the function may not work as expected or may not work at all. The DMSQSFSL CSL routine tells which level of the file pool server is being used. The application must know which parameters and functions are supported for the level of the file pool server.

Design Considerations

When designing your application program to manipulate files and directories, consider the following:

- Use a name definition, or namedef, to identify a file, directory or file mode letter to your program
- Acquire work unit IDs for work units
- Process SFS requests on behalf of other user IDs.
- Commit changes in application programs as you complete a task.
- Handle unexpected conditions in your program.

This section discusses these topics.

Using a Namedef

Many CSL routines require file names and file types, directory identifiers, file mode letters or some combination of these in their parameter lists. For any of these routines, you can specify a namedef instead. A namedef is a 1- to 16-character string that represents either:

- a file name and file type

- a directory ID or file mode letter.

By using namedefs in your parameter lists, you can run a program to process different files, file mode letters and directories without changing the code and recompiling the program. The file and directory names or file mode letters are defined externally to the program.

To associate a namedef with the name of a file, directory, or file mode letter, you issue a CREATE NAMEDEF command before running the program. The first character of the namedef must be alphabetic, and the remaining characters must be alphabetic or numeric. When the program runs, CMS does the operations on the file, directory, or file mode letter that the namedef represents.

You can use namedefs in three different ways depending on the operations that your program is going to perform.

- If your program uses different files within the same directory or accessed minidisk, you could use a namedef for the file name and file type and code the directory ID or file mode letter directly in the parameter list. This lets you process different files in the same directory.
- If your program uses the same file name and file type but different directories or accessed file mode letters, code the file name and file type directly but use a namedef for the directory ID or file mode letter. This lets you process different versions of the same file that reside in separate directories or accessed file modes.
- For the most flexibility, use namedefs for the file name, file type, and directory ID or file mode letter. In this case, the program can process any file name and file type as well as any directory ID or file mode letter.

Because a namedef is resolved at run time, you do not need to have them defined before compiling your program. If you run a program without defining the namedef, however, the program may fail because the namedef is not defined.

A namedef continues until the end of the CMS session (or until an abend occurs) unless you change the definition of the namedef or delete it altogether.

Creating a Namedef

Suppose you code your parameter list using the namedef FNAME for the file name and file type and DIRNAME for the directory ID. To process the file TEMP DATA in the directory POOLA:MARK.SURVEY, the user would have to enter two CREATE NAMEDEF commands before running your program:

```
create namedef temp data fname
create namedef poola:mark.survey dirname
```

If the file TEMP DATA was on an accessed file mode, (B for example), and could be either a minidisk or SFS directory, you could create a namedef for the file mode letter by issuing:

```
create namedef b fmname
```

To change the definition, you would enter a CREATE NAMEDEF command with a REPLACE option. For example, to change FNAME to refer to the file MYFILE DATA, you would enter:

```
create namedef myfile data fname (replace
```

See the sample program in Appendix F, “REXX Examples,” on page 545 for an example of using namedefs. See *z/VM: CMS Commands and Utilities Reference* for details on the CREATE NAMEDEF command.

Deleting a Namedef

To delete the namedef, enter a DELETE NAMEDEF command:

```
delete namedef fname
```

To delete all namedefs you have defined in this CMS session, enter:

```
delete namedef *
```

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

Additional Considerations for Directory ID

Many SFS commands and routines require directory identifiers as a part of their syntax and parameter lists. CMS allows the user or application to let the user ID and the file pool ID default. For example, in the command ACCESS `filepool:.B`, the user ID has not been specified. Before sending the request to the SFS server for this file pool, CMS fills in the user ID. This has several implications for the way your application program manipulates SFS files and directories:

Distributed Environments: When using SFS in a distributed environment, the user ID known locally may be different than the user ID known in the remote SFS file pool server. The virtual machine ID, which is the default, may not be the correct user ID.

Alternate User ID Support (DIAGNOSE X'D4'): In an application that uses DIAGNOSE X'D4' support, the application can imitate or act on behalf of an end-user who is not the user ID where the application is executing. Similar to the distributed environment case, CMS will fill in the default.

Multiple User ID Support: Applications making use of the multiple user ID support with the DMSGETWU routine have the same problem as alternate user ID support when they use the default user ID.

The CMS SET FILESPACE command allows the application to specify the correct user ID. For example, if an application uses DIAGNOSE X'D4' support to act as the user ID, MYNAME, the application could then issue 'SET FILESPACE MYNAME'. Any commands or CSL routines that follow could use the default user ID.

Note: If no CMS SET FILESPACE command was issued, or if it was issued but without any operands, the default user ID will be set to the virtual machine ID.

Use the CMS QUERY FILESPACE command to query the current default file space ID.

For more information on the CMS SET FILESPACE and CMS QUERY FILESPACE commands, see the *z/VM: CMS Commands and Utilities Reference*.

Using Work Units in Application Programs

A ***work unit*** identifies related SFS server requests that can be committed or rolled back concurrently. Some of the server requests identified with a work unit may make changes to one or more files or directories. These changes are what actually need to be committed. The work unit serves as a vehicle to ensure that all the changes are committed (or rolled back, if all the changes cannot be committed) in unison.

The changes identified by a work unit can be viewed as a ***logical unit of work***, a set of related actions whose results (changes) are to be treated as a single update. Only one set of related changes (logical unit of work) can be identified with a particular work unit at one time. When we refer to issuing a commit on a work unit, we are talking about committing the logical unit of work (the set of changes associated by that work unit). Once a commit has been done on the work unit, it can be used to associate another set of changes. It is rather like a wagon that can hold only one project at a time. Once the project is finished, you take it out of the wagon (commit the work), and then you can put another project into it to work on.

A work unit is identified by a fullword number called a ***work unit ID***. An application can have many active work units at any given time. Data can be committed whenever appropriate for your application. The files and directories associated with a particular work unit need not be closed in order for changes to be committed.

A work unit can be associated with multiple SFS file pool servers or multiple work units can be associated with one SFS file pool server. In addition, other resource managers can be accessed on the same work unit as SFS file pool servers.

A work unit can also be associated with a specific user ID, by using the DMSGETWU CSL routine. A service virtual machine with administration authority can issue file pool requests on behalf of disconnected user

IDs, or user IDs that differ from the VM ID of the service virtual machine, and all SFS authorizations will remain in force.

When you issue a COMMIT, CMS, through its Coordinated Resource Recovery (CRR), coordinates the commit of all protected resources accessed on the work unit. ***Protected resources*** are resources that conform to a set of requirements for participating in CRR. For more information on what CRR is and how it works, see Chapter 16, "Your Applications and Data Integrity," on page 241.

Use the routines in the following table to manage work units:

Table 6. Routines for Managing Work Units

CSL Call	Function	Description
DMSGETWU	Get Work Unit ID	Obtains a unique work unit ID if the default is not to be used.
DMSPOPWU	Pop Default Work Unit ID	Removes the latest default work unit ID from the work unit ID stack.
DMSPURWU	Purge Work Unit ID	Returns all work unit IDs to CMS; therefore, ends communication to all file pools.
DMSPUSWU	Push Default Work Unit ID	Identifies a specified work unit ID as the default work unit ID that will be used whenever a work unit ID is not specified. Puts default work unit ID on the work unit ID stack.
DMSQWUID	Query Work Unit ID	Returns the current default work unit identifier.
DMSRETWU	Return Work Unit ID	Returns to CMS the specified work unit. SFS files and directories open under the work unit are closed and an attempt is made to commit any outstanding work on the work unit.

When you IPL CMS, a default work unit ID is defined that will be used if you do not explicitly obtain one. You may want to obtain unique work unit IDs to:

- Separate the commit of data within your program
- Allow your program to call other programs or CMS commands and have those operations execute independently of your program.

Each work unit is independent of the other. That is, the changes made to files in a given work unit can be committed or rolled back independent of any other work unit.

When you enter CMS subset mode, CMS defines a new default work unit ID. When returning from subset mode, CMS returns the work unit(s) for the application. In other words, all local open files and directories are closed, a commit is done for all subset work units, and end-of-work unit processing is performed for all subset work units. Note that work unit IDs obtained outside of subset mode cannot be used in subset mode, and work unit IDs obtained in subset mode cannot be used outside of subset mode.

When files on minidisks get committed, it is not related to the COMMIT and NOCOMMIT parameters on CSL routines that operate on minidisk files. The work unit is part of the CSL interface, but it only applies to work being done to the Shared File System. Many CSL routines that operate on minidisk and SFS files (such as DMSCLOSE) require you to specify whether or not you want to commit the changes to the work unit or leave them in an uncommitted state (NOCOMMIT). When operating on minidisk files, the COMMIT and NOCOMMIT parameters refer solely to the work unit and have no effect on when the minidisk gets committed. Work done to a minidisk is not associated with a work unit.

For example, if you create a minidisk file using DMSOPEN and DMSWRITE, you must close the file using DMSCLOSE. The file will be committed when the close is completed (unless there are other files on that minidisk that remain open for output). If COMMIT was specified on DMSCLOSE, all uncommitted work associated with the work unit that was specified when the minidisk file was opened (DMSOPEN), will then be committed. If NOCOMMIT was specified, the work on the work unit will remain unchanged. See the explanation for DMSCLOSE in the *z/VM: CMS Callable Services Reference* for more details on closing SFS and minidisk files.

The same is true when using the CSL routines that allow you to commit and roll back work units. See "Committing and Rolling Back Changes in Application Programs" on page 141 for more details.

Obtaining Work Unit IDs

To get a work unit ID that is unique within a virtual machine you must use the DMSGETWU routine (Get Work unit ID). In the *workunitid* parameter you pass a 4-byte numeric field in which you want the work unit ID to be placed. It is this 4-byte field that you pass to other SFS program functions when you want to use a specific work unit ID. Work unit IDs obtained with DMSGETWU persist for the duration of the program.

When you enter CMS subset mode, CMS defines a new default work unit ID. If no unique work unit IDs are available, the *retcode* parameter contains a return code of 8. When you exit from CMS subset mode, all work units defined in subset mode are returned to CMS.

Example: To obtain a unique work unit ID, code:

```
CALL DMSCSL (DMSGETWU, RETCODE, REASCODE, WORKUNITID)
```

Returning the Work Unit ID

The DMSRETWU routine (Return Work Unit ID) returns a work unit ID to CMS that was previously obtained with the DMSGETWU routine. Returning the work unit ID informs CMS that your application has completed all work on the specified work unit and that the system can commit all outstanding work and free its storage associated with that work unit. You should explicitly commit all work before returning the work unit. DMSRETWU does not check or modify the work unit stack. Therefore, you should use the DMSPOPWU (Pop Default Work Unit ID) routine to remove the work units from the stack before returning them.

You need to return only those work unit IDs obtained by DMSGETWU. DMSRETWU closes all SFS files and directories that are open under the work unit and commits any outstanding work on the work unit. For more information on committing work, see "Committing and Rolling Back Changes in Application Programs" on page 141.

The *workunitid* parameter is a 4-byte numeric field for specifying the work unit ID.

Example: To return the work unit ID previously obtained using DMSGETWU, code:

```
CALL DMSCSL (DMSRETWU, RETCODE, REASCODE, WORKUNITID)
```

Changing the Default Work Unit ID

You can change the default work unit ID for a CMS session using the DMSPUSWU routine (Push Default Work Unit ID) and DMSPOPWU routine (Pop Default Work Unit ID). After you have obtained a work unit ID, use DMSPUSWU to place it on the work unit ID stack and identify it as the default work unit ID for the CMS session. The new default will be used whenever a work unit ID is not specified. You can only push those work unit IDs that you have obtained using DMSGETWU; you cannot push the system-generated default work unit ID.

You can find out what the current default work unit ID is by issuing the DMSQWUID (Query Work Unit ID) routine.

The DMSPOPWU routine (Pop Default Work Unit ID) removes a work unit ID from the work unit ID stack. The top default work unit ID is removed from the stack and the next one becomes the active default. You can specify the ALL parameter to remove all work unit IDs except for the CMS default one. In CMS Subset mode, only those work unit IDs obtained in Subset mode are removed. You should exercise care when using the ALL parameter. A called program, or subprogram, may have placed a work unit ID on the stack, planning to use it again at a later time.

Example: The following example obtains a work unit ID, places it on the work unit ID stack as the new default, and then later removes it from the work unit ID stack.

```
CALL DMSCSL (DMSGETWU, RETCODE, REASCODE, WORKUNITID)
CALL DMSCSL (DMSPUSWU, RETCODE, REASCODE, WORKUNITID)
/* The work unit ID in WORKUNITID is the new default */
⋮
/* To find out if the work unit ID you want is on the   */
/* top of the stack.                                    */
CALL DMSCSL (DMSQWUID, RETCODE, REASCODE, WORKUNITID)
⋮
/* Remove the work unit ID from the work unit ID stack. */
/* The previous work unit ID becomes the default.       */
CALL DMSCSL (DMSPOPWU, RETCODE, REASCODE)
/* Return the work unit ID. */
CALL DMSCSL (DMSRETWU, RETCODE, REASCODE, WORKUNITID)
```

Using Multiple Work Units in a Program

To achieve greater system performance and throughput, a program should open SFS files or directories at the last possible moment, and should commit the changes and close the files as soon as possible. By doing so, the program uses the least amount of SFS file pool server machine resources and helps keep shared files and directories available for other users.

To have greater control over how and when work is committed, you can use several concurrent work units. That is, you can have more than one work unit in process at a time. The programming interface lets applications use multiple work units to separate the commit of data, or for a called program to be independent of the calling program. This can be done by obtaining more than one work unit ID. You can group routine calls together by specifying the appropriate work unit IDs in the parameter lists of the routine.

Each work unit is independent of the other. That is, the changes made to SFS files in a given work unit can be committed or rolled back independent of any other work unit. For example, suppose your program does the following:

```
WORK1 = buffer for first work unit ID
WORK2 = buffer for second work unit ID

CALL DMSCSL (DMSGETWU, RETCODE, REASCODE, WORK1)
CALL DMSCSL (DMSGETWU, RETCODE, REASCODE, WORK2)

OPEN FILE1 using workunitid=WORK1
OPEN FILE2 using workunitid=WORK2
OPEN FILE3 using workunitid=WORK1
⋮
WRITE FILE1
WRITE FILE2
⋮
WRITE FILE3
⋮
CLOSE FILE2 using the COMMIT parameter to COMMIT work unit WORK2
⋮
CLOSE FILE1
⋮
----- System Failure -----
```

The changes to FILE2 were committed before the system failure, so even though the application did not complete, FILE2 was, in fact, changed. FILE1 is not changed because, even though it was closed, the system failed before the changes were committed. (This example assumes that FILE1 is recoverable).

If the COMMIT parameter was used when FILE1 was closed, the changes made to FILE1 and FILE3 would be made. Changes to both files would be made because they are on the same work unit. However, FILE3 would not be closed.

CMS resets the work unit environment at normal end-of-command or if an abend occurs. This means that any work units obtained using the DMSGETWU routine are no longer available.

Issuing CMS Commands in a Work Unit

So far we have discussed work units as they are used in CSL routines. There are, however, ways to update files maintained by SFS without using the routines. For example, in high-level languages, you can execute CMS commands that operate on a file in the Shared File System by calling the routine DMSCCE to issue a command from an exec (you can issue the command directly from REXX). In assembler, you can execute CMS commands by issuing a macro such as CMSCALL or SUBCOM, or by issuing SVC 202 or SVC 204. There is no way to specify a work unit when issuing CMS commands from a program or exec directly, so SFS uses the default work unit. You can, however, alter the default work unit with DMSGETWU, DMSPUSWU, DMSPOPWU, and DMSRETWU and, thereby, indirectly specify the work unit for CMS commands.

The default work unit for CMS commands issued from programs and execs is the same as that used for CSL routines. (XEDIT and session services use their own work units for output files.) This assures updates are committed. Using the same default has several important implications for your programs.

Suppose you have written a conversational application that updates files in the SFS file pool based on responses to your prompts. Most of the time your users respond normally to the prompts, and you continue to update the file. You did, however, decide to allow them to issue one of a subset of CMS commands in response to any prompt. This allows the user to issue ad hoc CMS commands without having to end the application. Here, in pseudo-code, is how it might be done:

```
Use DMSOPEN routine to open OUTFILE for output using default work unit ID
DO FOREVER:
     prompt user for input
     parse response
     is response a CMS command we can execute?
     Yes
       Issue the command
     No
       If a valid response
          Use DMSWRITE routine to write an appropriate record to OUTFILE
       else
         If end-of-conversation indicator
          Use DMSCLOSE routine to close OUTFILE with the COMMIT option
          end program
         else
          write message
         endif
       endif
END DO;
```

Suppose the user enters a CMS command. The program decides if it is valid and, if so, issues the command. If the command does not operate on a file in a SFS file pool, it does not matter to the program whether it succeeds or fails (unless it fails so dismally that it brings the program down with it). If it succeeds, the processing of OUTFILE is unaffected. If it fails, the same is true. Because the command does not operate on a file in a SFS file pool, no work unit ID is assigned to it. SFS does not know or care about the command.

Now, suppose the CMS command does operate on a file in an SFS file pool. In this case, SFS uses a default work unit ID. By coincidence, you have allowed the OPEN OUTFILE to default as well. They both have the same work unit ID, and the success or failure of both are linked together. Typically, CMS commands perform a commit, even on errors. If the CMS command fails in such a way that you roll back (receive return code 31) the changes, all changes in the work unit are rolled back. This includes both the changes caused by the CMS command before it failed and the changes previously made to OUTFILE — not necessarily the desired result, especially if many changes were already made to OUTFILE.

This situation may be avoided in one of the following ways:

1. If you wish OUTFILE updates to be unaffected by the commits and roll backs caused by CMS command processing, but require the capability of rolling back OUTFILE updates explicitly, you must specify different work unit IDs for the files used directly by your application (OUTFILE in the previous example) and for those files accessed by CMS commands issued through an exec, macro, or SVC. Because a default work unit ID is always used for CMS commands issued from a program or exec, you should obtain a different work unit ID to use with the SFS routines. To fix the previous example, we would not

let the work unit ID default when we opened OUTFILE. Instead, we would obtain a work unit ID for the SFS routines by adding the following to the beginning of the program:

```
WORK1 = buffer for work unit ID

CALL DMSCSL (DMSGETWU, RETCODE, REASCODE, WORK1)
Use DMSOPEN routine to open OUTFILE for output using workunitid=WORK1
```

2. If do not wish updates to OUTFILE to be rolled back even if your application experiences a failure, you should define OUTFILE as a nonrecoverable file. Using this approach, commits and rollbacks of the work unit caused by both CMS command processing and by your program will cause OUTFILE updates to be committed (where possible). Using this approach, you may fix the previous example by recoding the OPEN statement to include the NORECOVER option.

   Note: There are a number of ways to define files with the attribute of NORECOVER. Some of them may be used in conjunction with compatibility interfaces (such as EXECIO and FSWRITE). See "Using the Recoverability and Overwrite Attributes" on page 142 for a further discussion of the recoverability attribute.

Calling Modules or Execs in a Work Unit

When an application invokes a MODULE file that resides in an SFS directory, CMS uses the same default work unit used for SFS routines and CMS commands. CMS also implicitly opens the file, reads the file, closes the file, and may issue a commit, depending on whether other files are open. If the commit is issued, any uncommitted changes made on the default work unit before invoking the module are also committed.

For example, suppose you have an application that makes changes to a file and then calls another module to process some information. Your application is also using the default work unit and is not obtaining a different work unit before calling the other module. If the module resides in an SFS directory, then as a result of reading the module, CMS implicitly commits the updates made by your application before calling the module.

One way to avoid these implicit system commits is to obtain and push a new work unit before calling the module. See "Obtaining Work Unit IDs" on page 135 for information on obtaining a new work unit. Also, if your application uses only SFS resources, you can specify explicit work units on CSL calls. For information on how to specify these work units, see the description of the individual CSL routines in the *z/VM: CMS Callable Services Reference*.

When an application invokes an exec that resides in an SFS directory, CMS does **not** use the same default work unit. Exec processing uses a separate work unit to read an exec that resides in an SFS directory. Therefore, if your application invokes an exec that resides in an SFS directory, any uncommitted changes made by your application prior to calling the exec will not be committed.

Other Uses of Work Units

Although this chapter describes work units as they are used by SFS, work units may be used by other applications as well. A work unit is really just a token that is unique within a virtual machine. Work units are, however, intended to define a ***commit scope*** for program operations. That is, all operations associated with a single work unit should be able to be committed or rolled back as a unit. So long as your application adheres to that intent, the application can use work units for its own work.

Suppose, for example, you have an application that accesses a database. The database manager could associate related database operations just as actions taken by SFS CSL routines are associated with work unit IDs. The database manager would use its equivalent of work units to keep track of which operations must succeed or fail as a unit. When the database manager does its equivalent of a commit operation, all associated changes would be made permanent in the database.

Suppose that same application also updates SFS files using the CSL routines supplied with z/VM. If the database manager participates in CRR, changes on a work unit to both the database and the SFS files are coordinated. In other words, whether you use DMSCOMM or the database's equivalent of a commit, all

changes to both resources will be committed in unison for the work unit. For more information on CRR, see Chapter 16, “Your Applications and Data Integrity,” on page 241.

If the database does not participate in CRR, CMS cannot guarantee that updates to both resources will be committed simultaneously. The reason for this is that when updates are made to a nonparticipating resource on a work unit containing protected resources, CMS does not count the nonparticipating resource as a part of the work unit. Therefore, it is possible for the protected resources to be committed while the nonparticipating resource is left unchanged. You must issue a commit for that resource using the product-specific commit verb. To reduce problems, code your application to issue a resource-specific commit **first** to make the changes permanent to resources not participating in CRR. Then, if that commit succeeds, issue a coordinated commit for the protected resources. A commit verb of any protected resource causes a coordinated commit.

If you intend the SFS file pool and database operations to be separate, unrelated units of work, it is strongly recommended that you use different work unit IDs for those operations. If you do not, your application is not adhering to the intended use of CMS work units.

Atomic Requests

An atomic request is an SFS command or program function (CSL routine) that has immediate results. That is, any file pool updates that occur as a result of an atomic request are committed by the time the file pool server has finished processing the request.

Atomic requests do not participate in, or interfere with, Coordinated Resource Recovery. To keep atomic requests from interfering with coordinated work, SFS enforces the following rule:

> Do not issue an atomic request if there is outstanding (that is, uncommitted) work in the affected file pool for the specified (or default) work unit. If an atomic request is issued under this condition, the request is rejected with the appropriate return and reason codes.

Outstanding work in other SFS file pools (or other non-SFS resources) for the same work unit does not prevent an atomic request from being issued. The atomic request has no effect on the outstanding work for the other resources. Conversely, issuing DMSCOMM (Commit) or DMSROLLB (Rollback) while an atomic request is outstanding has no effect on the atomic request.

Note: An atomic request is "outstanding" only if the particular request supports and uses asynchronous communication by specifying the *requestid* parameter. This is the only case where the application can get control before the atomic request is complete. Again, issuing DMSCOMM or DMSROLLB at this point does not affect the atomic request. A subsequent check request for the outstanding asynchronous request will succeed.

The following is a list of atomic commands and routines for general use:

Table 7. General Use Atomic Commands and Routines

Commands	Routines	Task
CREATE LOCK	DMSCRLOC	Creates Lock
DELETE LOCK	DMSDELOC	Deletes Lock
DIRATTR	DMSDIRAT	Sets Directory Attributes
FILEATTR	DMSCATTR	Changes File Attributes
QUERY FILEPOOL DISABLE	DMSQFPDS	Querys a file space to determine if the file space or its owning storage group is disabled
QUERY LIMITS	DMSQLIMU	Displays Your Storage Space Limits
RELOCATE	DMSRELOC	Relocates Files/Directories
	DMSUDATA	Passes Information to an External Security Manager
QUERY ACCESSORS		Displays Information on Accessors of Directory Control Directories

Table 7. General Use Atomic Commands and Routines (continued)

Commands	Routines	Task
QUERY ENROLL		Displays Enrolled Users
SET THRESHOLD		Sets File Space Threshold Value

The following is a list of atomic commands and routines that require administration authority or that are intended for restricted use.

Table 8. Administration Atomic Commands and Routines

Commands	Routines	Task
DELETE USER	DMSDEUSR	Deletes User
ENROLL USER	DMSENUSR	Enrolls User
	DMSQLIMA	Displays User File Pool Information
	DMSRELBK	Releases Blocks
	DMSWRACC	Writes Accounting
DATASPACE		Makes a Directory Eligible For Use In a Data Space
DELETE ADMINISTRATOR		Deletes Administration Authority
DELETE PUBLIC		Deletes Connect Authority on All Users
ENROLL ADMINISTRATOR		Grants Administration Authority
ENROLL PUBLIC		Grants Connect Authority to All Users
FILEPOOL ENABLE	DMSENAFS DMSENASG	Enables use of a file space or storage group Enables Storage Group
FILEPOOL DISABLE	DMSDISFS DMSDISSG	Disables use of a file space or storage group Disables Storage Group
FILEPOOL RENAME		Renames the file space of an SFS user
MODIFY USER		Changes Space Allocation
QUERY ACCESSORS		Displays Information on Accessors of Directory Control Directories in Data Spaces
QUERY DATASPACE		Displays the Directory Control Directories Eligible for Data Space Use
QUERY FILEPOOL AGENT		Displays Information on Users or Internal Processes Running on an SFS File Pool Server
QUERY FILEPOOL CATALOG		Displays Information on File Pool Catalog Space
QUERY FILEPOOL COUNTER		Displays Information on SFS File Pool and CRR Recovery Server Counters
QUERY FILEPOOL CRR		Displays Information on CRR Recovery Server Counters
QUERY FILEPOOL DISABLE	DMSQFPDS	Queries a storage group, a file space, or all file spaces and all storage groups in a file pool to determine if they have been previously disabled
QUERY FILEPOOL LOG		Displays Information on SFS Log Minidisks
QUERY FILEPOOL MINIDISK		Displays Information on File Pool Minidisks

Table 8. Administration Atomic Commands and Routines (continued)

Commands	Routines	Task
QUERY FILEPOOL OVERVIEW		Displays Overview Information on a File Pool
QUERY FILEPOOL REPORT		Displays Information on an SFS File Pool and Server, and CRR Recovery Server
QUERY FILEPOOL STORGRP		Displays Information on Storage Groups
QUERY FILEPOOL STATUS		Displays File Pool Information

Committing and Rolling Back Changes in Application Programs

When you complete a work unit, you need to tell CMS what to do with the changes you made on the work unit. The changes can be saved, or *committed*, or they can be discarded, or *rolled back*. There are several ways to commit changes made in programs:

- Supply a COMMIT parameter in the parameter list of a CSL routine that operates on the work unit.
- Call the DMSCOMM (Commit) CSL routine.
- Call the SRRCMIT (Commit) SAA resource recovery (also known as CPI resource recovery) routine.
- Execute a commit verb of a resource that participates in CRR.
- Invoke the FSCLOSE macro. FSCLOSE can cause an implicit commit. For more information on FSCLOSE, see the *z/VM: CMS Application Development Guide for Assembler*.

To roll back changes made in programs, either:

- Call the DMSROLLB (Rollback) CSL routine
- Call the SRRBACK (Backout) SAA resource recovery routine
- Issue a rollback verb of a resource that participates in CRR.

Note: Changes to minidisk files can not be explicitly committed or rolled back. Changes are committed when the last file on the minidisk that is opened for output is closed. See Chapter 16, “Your Applications and Data Integrity,” on page 241 for details of how CRR coordinates commits and rollbacks for protected resources (those that participate in CRR).

It is important to commit your work even if you have not made any changes. For example, suppose you open a file, read a few records, and then close the file. You should commit your work because the work unit is still active, and subsequent atomic requests on the same SFS file pool and work unit ID will fail. Most often, the atomic routine issues a return code of 8 for work units left active. Committing also frees SFS file pool server resources when the accessed files have been previously closed.

If your application uses high-level language statements to update data controlled by a resource that participates in CRR, be sure to flush all data from the buffers (for example, close the files) before issuing a commit. This may be required because some high-level languages and assembler programs use OS/MVS queued sequential access method (QSAM) for output files. 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 CSL routines, FS macros, or the EXECIO command to write to SFS files avoids this situation.

Committing Your Work When You Have Exceeded a File Space Limit

During the course of your application, you may have written more blocks to a file space than are allocated. If you attempt to commit the work unit explicitly using SFS routines when the file space limit is exceeded, the commit will fail, but the work unit will not be rolled back. This is a situation called a *state check*. For example, if you have issued the DMSCOMM CSL routine to commit your work unit, a state check will be reflected back to you by a return code of 8 and reason code of 79061.

Your application must either delete unneeded file blocks that are above the file space limit and attempt to commit the work again, or explicitly roll back the changes if there is not sufficient space to hold them.

Using the COMMIT Parameter in SFS Routines

When you specify the COMMIT parameter as input to an SFS routine, (use DMSCLOSE as an example), you are, in effect, requesting two separate actions. Thus, there are situations in which you may be able to successfully perform the initial action of closing the file but unable to commit the work unit. This may be reflected as a state check, which implies that the work unit has been neither committed nor rolled back.

Your application can detect that a state check has occurred by checking for a return code of 8 (ERROR) and a reason code of 50500 after an SFS routine has been issued with the COMMIT parameter.

If you are only writing to a single file pool and your application does not use non-SFS CRR protected resources, a state check will only occur if you have exceeded your file space limit. In this case, your application may attempt to recover by erasing temporary files and attempting to commit the work unit, or by rolling back the work unit.

If you are using multiple file pools or CRR protected resources, there are a variety of errors you may encounter when attempting to commit the work unit. Your application must examine FPERROR data to further diagnose the cause of the COMMIT error. If there is sufficient space available in WUERROR, FPERROR will reflect all SFS error information. This will include an FPERROR block that has been created as follows:

- FPERETCD is set to 8
- FPEREAS is set to 50500
- FPEAUGMT is set to the reason code for the commit failure documented for the DMSCOMM routine (see *z/VM: CMS Callable Services Reference* for details).

Because multiple FPERROR blocks may be created, the cause of the state check may not be available unless the WUERROR parameter was coded such that it could contain more than one FPERROR block.

Note that if your application uses non-SFS protected resources, WUERROR will not return error information specific to those resources. Use the DMSGETSP CSL routine to retrieve all error blocks for all errors since the start of the last commit or rollback for the work unit.

Using the Recoverability and Overwrite Attributes

The recoverability and overwrite attributes of a file effect how a file is rolled back, updated, and recovered.

There are two extended file attributes that you can manipulate to control your program files. These two attributes are:

- Recoverability
- Overwrite.

These attributes apply to both minidisk and SFS files. In the case of minidisk files, the rules that govern these attributes are more restrictive.

Recoverability

The recoverability file attribute has two states, RECOVER and NORECOVER.

RECOVER
means that the file is recoverable in the sense of CMS commit and rollback support. A rollback to a recoverable file backs out all updates made since the last commit. RECOVER is the system default recoverability attribute.

NORECOVER
means that the file is nonrecoverable, in that it is not subject to CMS rollback support. Generally, a rollback causes updates to nonrecoverable files to be committed. However, in the event of an

abnormal termination of either the user machine or server, some updates to the file may be lost. Since minidisk files are not associated with a work unit, all minidisk files are considered as nonrecoverable.

For example, you may want to use nonrecoverable files if you have data, such as transaction logging data, that you want committed in the event of an ABEND. You can then keep track of any failing transaction.

Overwrite

The overwrite attribute can have two states: NOTINPLACE and INPLACE.

NOTINPLACE
means that readers of the file see a consistent version of the file between open and close, because file writes are shadowed. NOTINPLACE is the system default overwrite attribute.

INPLACE
means that read consistency is not required and updates to existing blocks of the file do not always require additional blocks to be allocated to store the modified data. With the INPLACE attribute, readers of the file might not see a consistent version of the file between open and close, and may see updates that have not yet been committed. For minidisk files, file mode 6 files have the INPLACE attribute.

There are a set of integrity exposures associated with the INPLACE file support. In general, they are similar to the exposures experienced with file mode 6 files on minidisks:

- Any exception condition that might occur during the writing process can cause only part of the data to be written to the file if:
  - The record being written is larger than a CMS block (4KB).
  - The record being written spans a CMS block. This could happen for any LRECL if records span blocks.
- Even if individual records are not exposed to partial writes, file integrity may still be compromised if only a subset of the records written is actually updated in the SFS file pool. This is particularly dangerous for an application where the meaning of one record is dependent on the value contained in another record.

You may use the FORCE option of DMSWRITE or frequent COMMITs by the writer or both, in conjunction with the REFRESH option of DMSREAD, to control when updates to an SFS INPLACE file are seen by a reader. This enables one or more readers to access the data as it is being written to an SFS INPLACE file. For more information see, “Seeing Uncommitted Updates” on page 145.

Manipulating Extended File Attributes

You can use the following CSL routines and CMS commands to create, change, or query the overwrite and recoverability file attributes for SFS files.

Table 9. CSL Routines

CSL Routine	Function	Description
DMSCATTR	Change Attributes	Modifies the recoverability and overwrite attributes of an SFS file.
DMSCRFIL	Create File	Creates a new empty file in an SFS directory. You can specify recoverability and overwrite attributes.
DMSEXIST	Exist	Checks on the existence of a file or directory and returns information about it (if it exists) in a buffer.
DMSGETDI	Get Directory	Reads directory records containing extended file attributes when opened with intent FILEEXT.
DMSGETDX	Get Directory - File Extended	Reads one directory record of extended file attributes.

Table 9. CSL Routines (continued)

CSL Routine	Function	Description
DMSPOPA	Pop Attributes	Removes a set of recoverability and overwrite attributes previously defined with DMSPUSHA.
DMSOPEN	Open	Opens a file for subsequent reading or writing. You can specify recoverability and overwrite attributes.
DMSPUSHA	Push Attributes	Defines a set of default recoverability and overwrite attributes for each file mode number.

Table 10. CMS Commands

CMS Command	CSL Routine	Description
CREATE FILE	DMSCRFIL	Creates a new empty file in an SFS directory.
FILEATTR	DMSCATTR	Modifies the recoverability and overwrite attributes of file(s) residing on an SFS directory.
QUERY FILEATTR	DMSGETDI or DMSGETDX	Retrieves the recoverability and overwrite attributes of a file residing on an SFS directory. The QUERY FILEATTR command also displays the attributes of a file. DMSGETDI returns the extended file attributes when the directory is opened with intent FILEEXT.

The routines are described in the *z/VM: CMS Callable Services Reference*. The commands are described in the *z/VM: CMS Commands and Utilities Reference*.

Committing SFS Changes in Application Programs

Supplying a commit parameter on a CSL routine has two distinct advantages. When you use commit parameters, you can attempt to request two functions using a single communication, which is much more efficient, and you can commit a group of related changes all at the same time. A typical approach would be to use the commit parameter when you are closing the last file or directory. The following is pseudo-code for a typical close and commit.

```
GET WORKUNITID unit1

OPEN FILEA DATA POOL1: for write using workunitid=UNIT1
OPEN FILEB DATA POOL1: for read using workunitid=UNIT1
OPEN FILEC DATA POOL1: for read using workunitid=UNIT1
.
CLOSE FILEA DATA POOL1:
CLOSE FILEB DATA POOL1:
CLOSE FILEC DATA POOL1: and COMMIT workunitid UNIT1
```

Executing a DMSCOMM routine can separate the commit from another routine. This is necessary if, for example, your program calls a subroutine that, because it is a general-purpose routine, does not issue its own commits. In this case, you may want to enter a DMSCOMM before continuing with your own program's work. For example,

```
OPEN FILEA DATA (let work unit ID default)
.
WRITE to FILEA DATA
.
COMMIT the default work unit
.
.
CALL other_prog          (subroutine that does not issue COMMITs)
COMMIT the default work unit
.
```

In this example, all work unit IDs are allowed to default. The main routine issues a commit before calling the subroutine while the file is open. The subroutine does some work using the default work unit ID, but does not issue a commit. On return, the caller issues a commit for the general purpose subroutine, then continues work. This result in a coordinated commit of all protected resources.

Although it may be necessary to issue separate DMSCOMM requests, for best performance, you should use the commit parameter instead of the DMSCOMM routine.

If an SFS file or directory is open when you use a commit parameter, the commit updates the open file or directory. Committing changes while the file or directory is open, lets you save updates at intervening points before you are done processing. For example:

```
OPEN FILEA DATA (let work unit ID default)
⋮
WRITE
⋮
COMMIT
⋮
WRITE
⋮
COMMIT
⋮
WRITE
⋮
CLOSE and COMMIT FILEA DATA
```

Although not recommended, yet another way to commit work is to let CMS implicitly commit the changes. That is, you use a NOCOMMIT parameter or you do not use a DMSCOMM routine (or other commit verb). In this case, CMS automatically closes any open files and commits the changes upon successful end of command. This method also results in a coordinated commit of all protected resources.

End of command can be thought of as the point at which you see the CMS Ready; message. If you create a MODULE for a program and then invoke that program from the CMS command line, the program executes, returns control to CMS, and you see a Ready; message. In this case, the end of program is also the end of command.

Suppose, on the other hand, 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. Furthermore, if your program does not commit its work, it could prevent atomic requests from executing in the exec that calls your program. However, when an application invokes a MODULE file that resides in an SFS directory, CMS implicitly issues a commit as a result of reading the module. See “Calling Modules or Execs in a Work Unit” on page 138 for more information about invoking a MODULE file.

The problem is that you cannot always predict how users run your program. While you may want it to run stand-alone, there is no guarantee that a user will not put it in an exec with other programs or call it using some other mechanism. It is recommended that you close your files and commit your work units rather than relying on the system to commit for you.

Seeing Uncommitted Updates

Your program may or may not see uncommitted updates to the SFS file pool, depending on whether the program files are INPLACE or NOTINPLACE. Some rules to remember are:

1. If your program files have the NOTINPLACE attribute, other users see only your committed updates. (NOTINPLACE is the default setting of the overwrite attribute if NOTINPLACE or INPLACE is not used on the DMSOPEN routine or if the DMSPUSHA routine was not issued.) If the file resides in a file control directory, users will not see the changes until they close the file and reopen it. If the file resides in a directory control directory, the access status of the directory determines, in part, when a user sees the data. When the directory is not accessed (the user is directly referencing the file using CSL routines), the user does not see your committed changes until he or she closes and reopens the file. If, however, the user has the directory accessed, he or she will not see the changes until the directory is reaccessed.

2. If the files have the INPLACE attribute, readers might not see a consistent version of the file between open and close. Again, the rules vary slightly for files residing in file control and directory control directories. For file control directories, users can see updates to existing records as soon as they are written, regardless of whether you committed the data. They cannot see extensions to the file unless they close and reopen the file.
3. Users can also see uncommitted updates to existing records in INPLACE files in directory control directories. If the directory control directory is accessed, however, they can see file extensions only if they reaccess the directory. If the directory is not accessed, they see any extensions when they close and reopen the file.
4. In all cases, users can read your updates to existing records (committed or uncommitted) in SFS INPLACE files as soon as they are written to the file pool. Due to the cache that CMS maintains in your virtual machine, however, changes may not be immediately transmitted to or from the file pool server. That is, changed records may be momentarily held in the writer's cache while existing records in the reader's cache may prevent CMS from retrieving current records from the file pool.

   If it is important to your application to have changed records immediately written and the latest records read, use the FORCE option on DMSWRITE and the REFRESH option on DMSREAD. This circumvents the use of the cache. While performance is impaired when the cache is not used, you have greater control of your data.
5. You can commit while your file is open, regardless of whether your file is INPLACE or NOTINPLACE.
6. You can see uncommitted updates for your own work when you close the file and then reopen it in the same work unit. If you open a NOTINPLACE file in a different work unit, you will not see the uncommitted updates. For INPLACE files, even if you open the file in a different work unit, you may see some updated portions of the file.
7. For NOTINPLACE files, you never see someone else's uncommitted updates, even if they closed the file.
8. For NOTINPLACE files, programs operating on one work unit cannot see uncommitted changes on another work unit.

Rule 3 is most interesting for your own programs. You can open a file, make changes to it, and close it. In the same work unit, you can reopen the file and read it to verify the changes (or write additional changes), then close and commit file. When reopening the file, it does not matter whether you use the name of the base file, or an alias for it, or a NAMEDEF that points to it. If you reopen the file in the same work unit, you see the uncommitted updates. The changes are not permanently made until you commit the work. You may run into the lock already held by the work unit making the changes.

If you open an INPLACE file with the intent of REPLACE, updates to the file are not written in place. They are shadowed.

After you commit the work unit, others can see the changes you made. Any user who opened the file before you committed the changes would see the old copy of the file. They would not see the new copy of the file unless they closed and reopened the file (regardless of whether they start a new work unit). The exception to this is if the file is update-in-place (has the INPLACE file attribute). With an INPLACE file, the user may see some updates as they occur. In addition, for SFS INPLACE files the REFRESH option of DMSREAD and the FORCE option of DMSWRITE lets you specify when INPLACE updates are made visible to readers.

Note that many SFS operations, such as granting authority or erasing a file, are not final until the work unit is committed. In addition, atomic functions, notably Create Lock (DMSCRLOC) and Delete Lock (DMSDELOC), are rejected if any uncommitted functions exist in the affected SFS file pool on the same work unit.

Rolling Back SFS Changes in an Application Program

Recoverable Files: Use the DMSROLLB routine (Rollback) to discard, or roll back, any changes made to recoverable files within a work unit. The changes may have been to one or more files or other objects, such as directories and subdirectories, in one or more SFS file pools or other resources that participate in CRR. For example, suppose your program writes to a file in each of two SFS file pools within a work

unit, and then it encounters an error. Depending on the severity of the error, you may want to roll back all the changes in the work unit to ensure the integrity of the data in the updated file or directory. You would issue the DMSROLLB routine similar to the following:

```
WORK1 = WORKUNITID

CALL DMSCSL (DMSROLLB, RETCODE, REASCODE, WORK1)
```

You have rolled back all the changes within the work unit. You can continue processing with the work unit ID if you want.

Rollbacks, like commits, are coordinated in that they discard all the changes to all protected resources modified on the work unit. As in the case of commits, coordinated rollbacks can also be issued by using:

- SRRBACK (Backout) or DMSROLLB (Rollback) routine
- Rollback verb from a resource that participates in CRR.

Note: A rollback closes all SFS files and directories in the work unit.

Nonrecoverable Files: Updates to files that have the NORECOVER attribute are not rolled back in the event of an application initiated rollback or abnormal application termination. As much data as possible is committed.

The permanent state of an updated NORECOVER file would be unpredictable following:

- Abnormal virtual machine terminations
- Abnormal communications link terminations
- Abnormal SFS server terminations.

In any of these cases, the file is readable, but some updates between the last commit and the abnormal termination may not be committed.

Handling Unexpected Conditions in SFS

When a program encounters an unexpected condition, it can do one of three things:

- Use DMSROLLB to roll back changes made to a file since the last commit. This will ensure the integrity of the data in the updated file or directory. Once you have rolled back all the changes, you can continue processing with the work unit ID.
- Terminate processing or set a return code or issue a message and return code to the user or all. For this case, the commit will be attempted at end of command (when the Ready; message is displayed).
- Use DMSPURWU with the FORCE option and end the program (or, if written in assembler, issue the DMSABN macro to terminate the program). In this case, a roll back will be issued. When writing your program, examine the available error information to decide what type of error handling routines you need to provide.

Collecting Error Information

The CSL routines for CMS file and directory manipulation provide three sources of error information:

- Return codes
- Reason codes
- Workunit extended error information.

The first two sources are required parameters for every routine; the last source is optional.

The ***return code*** provides general error information for a routine. The return code is placed in the return code variable that you provide and in general register 15. (The return code variable is a signed fullword.) SFS routines may have one of the following return codes:

0
Normal—the routine executed successfully.

4
Warning—the routine executed, but the result may or may not be as intended.

8
Error—If the commit parameter was specified on the routine call, a return code of 8, reason code of 50500 means that the routine did execute, but the commit was not performed. Otherwise a return code 8 means that the routine did not execute.

12
Error—the routine did not execute and work within the work unit ID was rolled back.

16
Severe Error—the commit was performed, but the state of coordinated resources may not be consistent.

20
Severe Error—the work was rolled back, but the state of coordinated resources may not be consistent.

Return codes that are greater than 8 can only be received when there is an error in an operation to the Shared File System or other CRR participant. Errors that apply to a minidisk will return an error code of 8.

You may also receive return codes from DMSCSL that are negative values. For more information on these return codes, see the *z/VM: CMS Callable Services Reference*.

For many conditions where you receive a return code of 8, the state of the work unit stays the same. For example, if you are in a work unit, issue a routine, and receive a return code of 8, you are still in the work unit, and you must commit or roll back any work before you issue an atomic request or before you try to write to another SFS file pool.

Return codes 4, 8, 12, 16, and 20 have ***reason codes*** associated with them to further describe the warning or error condition. The reason code for a routine is placed in the reason code variable that you provide and in general register 0. (The reason code variable is a signed fullword.) Return code 0 does have an associated reason code of 0. For a list of the return codes and associated reason codes, see the *z/VM: CMS Callable Services Reference*.

Workunit extended error information (*wuerror* parameter) is an optional source of error information. It contains additional error information from the Shared File System. No minidisk error information is contained in *wuerror*. On input, the *wuerror* parameter is a character string followed by a length parameter. If you omit it or if the variable has a length field with a value of 0, only the return code and reason code are returned. The following information is returned by *wuerror*:

- Length of the *wuerror* parameter
- Number of SFS file pool error information areas returned
- Total number of errors for which information is available
- One or more groups of SFS file pool error information.

The SFS file pool error information contains information about errors that have occurred. This information includes: error reason codes, warning reason codes (up to 16 possible), and user ID index. The routine, DMSWUERR, converts the *wuerror* output to data placed in individual variables. Macros WUERROR and FPERROR let routines map into the work unit (WUERROR) and SFS file pool (FPERROR) data areas. For more information, see the *z/VM: CMS Callable Services Reference* and the *z/VM: CMS Macros and Functions Reference*.

Abend Recovery

Your application can call the Purge Work Unit IDs (DMSPURWU) CSL routine with the FORCE option to perform some recovery operations for SFS and any other protected resources accessed on the work units. DMSPURWU performs the following SFS and CRR activity:

- Closes all SFS files and directories.
- Rolls back all uncommitted changes to recoverable SFS files and directories and to other protected resources accessed on all work units. (Rollback of nonrecoverable files may cause updates to be committed to DASD.)

- Deallocates all protected conversations.
- Returns all work unit IDs.
- Ends communications to all SFS file pools and releases all session length locks in the SFS file pools for the issuer.
- Clears all nondefault file attributes.

For details on the DMSPURWU (Purge Work Unit IDs) routine, see the *z/VM: CMS Callable Services Reference*.

An assembler program can issue the DMSABN macro (or the OS ABEND macro if using simulation) to perform recovery operations in a program that manipulates SFS files and directories (and any other protected resources). DMSABN performs the following SFS and CRR activity:

- Closes all files and directories. (This includes files on a minidisk and in an SFS file pool.)
- Rolls back all uncommitted changes to recoverable SFS files and directories and to other protected resources accessed on all work units. (Rollback of nonrecoverable files may cause updates to be committed to DASD.)
- Deallocates all protected conversations.
- Returns all work unit IDs.
- Clears all defined namedefs.
- Resets default file attributes.

For details on the DMSABN macro, see *z/VM: CMS Macros and Functions Reference*.

File I/O

This section describes how to use CSL routines in VMLIB to manage SFS and minidisk files. The routines that let you manage files are collectively known as File I/O routines. Here are some of the File I/O routines that you can use for file I/O:

Table 11. CSL Routines for File I/O

CSL Call	CSL Function	Description
DMSCATTR	Change Attributes	Modifies the recoverability and overwrite attributes of the specified SFS file.
DMSCLOSE	Close	Closes a file (logically disconnects an application program from a specific file).
DMSCLDBK	Close Blocks	Closes a file for reading and writing data blocks (logically disconnects an application program from a specific file).
DMSCOMM	Commit	Commits changes to the work unit.
DMSCRFIL	Create File	Creates a new, empty file in a SFS directory.
DMSERASE	Erase	Erases files (minidisk, base files and aliases) and directories.
DMSEXIFI	Exist - File	Checks for an existing file and return the file information in variables.
DMSEXIST	Exist	Checks for an existing file (or directory) and return the file information in a buffer.
DMSOPDBK	Open Data Blocks	Opens a file for reading and writing file data blocks (logically connects an application program to a specific file).
DMSOPEN	Open	Opens a file (logically connects an application program to a specific file).

Table 11. CSL Routines for File I/O (continued)

CSL Call	CSL Function	Description
DMSPOINT	Point	Alters the read and write record pointers in a file opened by DMSOPEN.
DMSPOPA	Pop Attributes	Removes the most recent recoverability and overwrite default attributes set by DMSPUSHA.
DMSPUSHA	Push Attributes	Defines a set of default attributes for each file mode number.
DMSRDDBK	Read Data Blocks	Reads one or more file data blocks.
DMSREAD	Read	Reads one or more records from a file.
DMSRELOC	Relocate	Moves a file, subdirectory or external object from one SFS directory to another.
DMSRENAM	Rename	Renames a file, subdirectory or external object.
DMSROLLB	Rollback	Rolls back uncommitted changes to recoverable SFS files changed in the work unit. The changes made since the start of the work unit are rolled back. Changes made to nonrecoverable SFS files are committed.
DMSTRUNC	Truncate	Truncates a file to a specified record.
DMSVALDT	Validate	Checks validity of a file identifier.
DMSWRDBK	Write Data Blocks	Writes one or more file data blocks.
DMSWRITE	Write	Writes one or more records to a file.

Using CSL Routines and Existing FS Macros

CSL routines that perform file I/O can work on both SFS and minidisk files. You can have programs that are accessing the same minidisk and SFS files using both the CSL routines and FS macros. You cannot mix calls between the two interfaces. For example, you cannot open a file using DMSOPEN and read it using FSREAD. You can open the file using FSOPEN and open it again using DMSOPEN. Each interface has it's own restrictions that apply. This is true for both minidisk and SFS files. Some of these restrictions are:

- A file (minidisk or SFS) may only be opened once using FSOPEN.
- Files may be opened multiple times 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).

Because there is no native file sharing capability for minidisk files, the file system must keep track of which minidisk files are being opened and what interface is being used. A minidisk file may only be opened for output once per virtual machine. If the FSOPEN macro is used to open the file for output, DMSOPEN can not open the file for output. If DMSOPEN opens the file for input, FSOPEN could still open the file for input also. In fact, DMSOPEN could open the file many times for input, but FSOPEN can only open it once. Once a minidisk file is opened for output, it must be closed before either interface can open it again.

The CSL interface allows applications to use work units to commit and roll back work for SFS files. When using minidisk files, no work is associated with a work unit. Minidisk files are committed when the last file that is opened for output on a given file mode is closed. Minidisk files are typically committed by using the FINIS command or FSCLOSE macro. This is considered to be part of the FS macro interface but does affect minidisk files that have been opened using CSL routines. If you close all files on a file mode using FINIS or FSCLOSE (FINIS * *):

- all files opened using FSOPEN are closed
- all files opened using DMSOPEN and DMSOPDBK are temporarily closed.

After this has completed, the minidisk files have been committed but the files opened by the CSL routines remain open. This allows applications that use FINIS to commit the minidisk changes regardless of the interface used to open the file. SFS files opened using the CSL routines will not be committed by using FSCLOSE or FINIS. The files opened using the CSL routines remain open so the FINIS invocation is transparent to these files. All files opened using the CSL routines must be closed explicitly by the appropriate CSL routine. There is no CSL routine that has the equivalent function of “FINIS * *”.

The most general rule that can be applied to using the CSL and FS macro interfaces is:

- For SFS files, the SFS rules of file sharing apply. A file may be opened once for output and multiple times for input. FSOPEN will only allow the file to be opened once.
- For minidisk files, the file can only be opened once for output by either interface. It can be opened multiple times for input by DMSOPEN and only once for FSOPEN.

Handling Files and Directories Opened Using File Mode

The file mode actually associated with a specific open file or directory may not be the file mode specified by your application. This can occur when the file mode you have specified has read-only extensions.

When File Mode is Associated with a Minidisk

Files and directories opened on minidisks are always associated with a specific file mode. When that file mode is no longer accessed, all files and directories related to that file mode will be closed and committed regardless of the interface by which they were opened.

When File Mode is Associated with a FILECONTROL Directory

Files and directories opened on SFS directories have been opened via either a file mode or directory id. Regardless of the mechanism used to identify the directory id, on SFS directories with the FILECONTROL attribute, only SFS files opened using FS macros will be closed when the file mode is released. The others will remain open until they are explicitly closed, or rolled back by either the application or system processing.

When File Mode is Associated with a DIRCONTROL Directory

Open files and directories on SFS directories with the DIRCONTROL attribute are handled differently. When the last accessed instance of a particular DIRCONTROL directory is released, all open files and directories open with the intent of FILE for that directory are closed and committed.

Determining If a File Exists

To determine if a file exists and the status of a file, use the DMSEXIFI or DMSEXIST routine. However, if this is an SFS file and you plan to do subsequent reads or writes to the file **and** share the file with other users, you should not bother with determining if the file exists. You should begin file I/O by opening the file with an intent other than READ. This assures that no other users could erase or revoke the file before you get a chance to open the file. Once you have the file opened, other users cannot erase the file.

DMSEXIFI returns file information in variables. If the file does not exist or you are not authorized to read from the directory, DMSEXIFI returns a return code of 8 as the value of the return code parameter and a reason code of 44000 as the value of the reason code parameter. If the file does exist, DMSEXIFI returns a return code of 0 and places the file information in the variables that you provide. Check the authority information passed back to make sure you have the correct authority. If the file resides on a minidisk, check the authority information to determine how the minidisk was accessed.

DMSEXIST returns file information in a specified buffer. If the file does not exist or you are not authorized to read from the directory, DMSEXIST returns a return code of 8 as the value of the return code parameter and a reason code of 90220 as the value of the reason code parameter. If the file does exist, DMSEXIST

returns a return code of 0 in the return code parameter and places the file information in the specified buffer. Check the authority information passed back to make sure you have the correct authority. If the file resides on a minidisk, check the authority information to determine how the minidisk was accessed.

You can map the output with the EXSBUFF assembler mapping macro. For more information on EXSBUFF, see the *z/VM: CMS Macros and Functions Reference*.

Creating Empty SFS Files

There are many ways to create empty SFS files. Some of these are discussed here, using the:

- XEDIT command
- CREATE FILE command
- ERASE command with the DATAONLY option
- DMSOPEN CSL routine
- DMSCRFIL CSL routine
- DMSOPEN CSL routine with the ALLOWEMPTY option.

Using the XEDIT command: If you enter

```
XEDIT fn ft fm
```

and the file identified by *fn ft fm* does not exist, a new empty file is created. You can then put some data in the file and SAVE or FILE it. You can also save an empty file, if the directory is in a file pool that supports empty files. However, you must use SSAVE or FFILE to write the empty file to the directory.

Using the CREATE FILE command: Entering

```
CREATE FILE fn ft dirid
```

creates an empty file in an SFS directory. CREATE FILE cannot be used to create files on minidisks.

Using the ERASE Command with the DATAONLY Option: Erases all the data in an SFS file without deleting the file from the directory. All existing authorizations and aliases are retained.

Using the DMSOPEN CSL routine: As described in "Opening Files" on page 152, if you open a file with intent NEW, a new file is created. Also, if you open a file with intent WRITE or REPLACE, a new file will be created if one by that name does not exist. You can explicitly specify that an empty file may be created by specifying the ALLOWEMPTY parameter with DMSOPEN or DMSOPDBK CSL routines. If you do not specify ALLOWEMPTY and you also do not write any records to the new file before closing it, no file will be created.

Using the DMSCRFIL CSL routine: DMSCRFIL creates a new empty file in an SFS directory.

Using DMSOPEN CSL Routine with the ALLOWEMPTY Option: Can create an empty SFS file if there are no records in the file when a DMSCLOSE or COMMIT is issued. Either a new empty file is created or an existing file is replaced with a empty file.

Opening Files

To open a file, that is, establish a logical connection to the file for subsequent reading or writing of records, or both, use DMSOPEN. If you want to perform data block I/O operations on a file, see "Data Block I/O" on page 163 for more details. The DMSOPEN routine lets you open a file and specify the intent, or the type of operation that you are performing, and the type of I/O that you want performed.

You can indicate the intent by specifying one of the following parameters:

- READ
- NEW
- REPLACE
- WRITE.

READ means that the file will only be read. You cannot open a file with READ if it does not exist.

NEW indicates that the file does not exist and it will be created. You can write to the new file or read from it. Use the F or V parameter to create a fixed- or variable-length record format; if not specified, the default is fixed (F).

REPLACE indicates that if the file exists, you will replace it with only the added records. If the file does not exist, it will be created. When you have opened a file for replace, you can only read records that you have written. Attempting to read records before writing any results in an end-of-file condition (return code = 4). If you close the file before writing any records, the records in the file before the open are available for a future open.

When an SFS file is replaced, the old version of the file is shadowed by the SFS server. If no records are written to the file or you do not want to actually replace it, this operation can be rolled back later on. For a minidisk file when REPLACE is specified, the file is erased when it is opened. If no records are written to the file, it will no longer exist.

Use the F or V parameter to create a fixed- or variable-length record format; if not specified, the default is fixed (F).

WRITE indicates that you will write to and read from the file. All changed and added records are written; other records remain unchanged. A new file is created if the specified one does not exist.

If the file exists, use the record format of the file when adding records. If the file does not exist, use the F or V parameter to create a fixed or variable-length record format; if not specified, the default is fixed (F).

You can also indicate the type of I/O you want to be performed when you open a file. Specify one of the following parameters:

- **CACHE** should be specified when the caller intends to read the data sequentially most of the time. Specifying this parameter causes the file system to cache several data blocks for the file, performing I/O only when the cache buffer is full (for writing) or empty (for reading). This generally reduces the number of separate I/O operations performed on the file.
- **NOCACHE** should be specified when the caller intends to read the data in random order most of the time.

If you do not specify CACHE or NOCACHE, the system chooses a method that it considers appropriate.

You may specify whether the file is recoverable or nonrecoverable. The RECOVER attribute specifies that uncommitted changes are backed out as the result of an application initiated rollback. NORECOVER specifies that changes are not rolled back by an application initiated rollback. Instead, a rollback causes the updates to be committed (in most cases). All minidisk files are considered nonrecoverable.

The overwrite attribute may also be specified. If the file is NOTINPLACE, writes are to be shadowed such that readers see a consistent version of the file from open to close. If the file is INPLACE, updates are made in place where possible, for reduced DASD utilization. If you have an INPLACE file open for READ, you may see some uncommitted file updates. However, you have to reopen the file to see extensions (new blocks and records) that have been written and committed to the file. For files residing in accessed directory control directories, you must reaccess the directory to see extensions. The overwrite attribute for minidisk files is determined by the file mode number. If the file mode number is 6, the file is treated as INPLACE. For all other file mode numbers, the file is treated as NOTINPLACE.

You cannot change file attributes using DMSOPEN unless you open with the intent of REPLACE.

The OPENRECOVER parameter can indicate that all updates to the file resulting from this open are treated as if the file had the attributes RECOVER and NOTINPLACE. This does not mean that these two attributes are stored with the file. Once the file is closed and committed, the changes to the file, and any subsequent activity to the file, are handled in a manner consistent with the attributes assigned to the file.

When you open a file, CMS passes a ***token*** back to your program. The token is an 8-byte field that identifies the file. You will pass this token on to other routines for reading, writing, and closing the same file.

After a file is open, you may need to determine certain attributes of the file, such as the number of records in the file, record length, record format, and the date and time the file was last modified. You can use

the extract function of the DMSERP routine to obtain this type of information. See the *z/VM: CMS Callable Services Reference* for more information on DMSERP and the extract functions that are available.

Example—Opening a File: The following REXX program uses the DMSOPEN routine to open, with the intent to read, the GETFILE EXEC file residing in SERVER8:FAIRLIEA.:

```
/*  Opens a file. */
retcode=0
reascode=0
fileid1.fname = 'GETFILE'
fileid1.ftype = 'EXEC'
fileid1.dirname = 'SERVER8:FAIRLIEA.'
fileid1 = fileid1.fname fileid1.ftype fileid1.dirname
fileid1len = length(fileid1)
opentype = 'READ'
opentypelen = length(opentype)

call csl 'DMSOPEN retcode reascode fileid1 fileid1len opentype',
         'opentypelen token'
exit
```

Reading and Writing Files

This section contains examples illustrating how to use the following CSL routines to read and write files. These routines are:

- DMSREAD -- Reads data sequentially or specifically by position
- DMSWRITE -- Writes data sequentially or specifically by position.

Reading Files

DMSREAD (Read) obtains one or more records from a file. The records can be fixed or variable, and they can be read sequentially or specifically by position in the file.

By default the first read operation begins with record number 1, and subsequent reads start at the next sequential record. In addition to the *retcode* and *reascode* parameters, specify the following parameters:

- *token*
- *records*
- *datalength*
- *buffer*
- *bytesread*.

The following parameters are optional:

- *position*
- *wuerror*
- REFRESH | NOREFRESH
- *requestid*.

Token

Token, returned on a previous DMSOPEN, identifies the file to be read.

Records

Records specifies the number of records to be read.

Datalength

Datalength specifies the maximum number of bytes to be read. The value must be less than or equal to the size of *buffer* in bytes.

To avoid a truncation warning, *datalength* must be specified, for fixed-length records, as the product of the number of records to be read and the logical record length (lrecl):

datalength=records x *lrecl*

However, it is not considered an error if *datalength* is not equal to this product. When it is not, it is possible to skip data in the file during sequential reads. For example, suppose a file has twenty 80-byte, fixed-length records. If the first read from the file requests five records (5 x 80 = 400) and specifies a *datalength* of 300, the data from the first three records (3 x 80 = 240) and the first 60 bytes of the fourth record are placed in the buffer (240 + 60 = 300), and a truncation warning is returned (return code = 4). If the next read does not specify a position number, reading begins with record six, thereby skipping the last 20 bytes in record four and all of record five.

For variable-length records, the datalength is equal to the maximum logical record length, or

datalength = 1 x *lrecl*

Buffer

Buffer specifies the area into which data will be placed.

Bytesread

After the read operation, *bytesread* is set to the number of bytes of data actually placed in the buffer. It is set to 0 if the read fails with a return code of 8 or 12.

Position

Position specifies the number of the next record to be read relative to the beginning of the file (record 1). The next sequential record is read if position is not specified or if 0 is specified (0 is the default).

An end-of-file warning (return code of 4) is returned only when no data is placed in the buffer.

CMS supports sparse files for fixed-length records only. A sparse file is a file with missing, or skipped, records. If you attempt to read a record that has never been written, CMS returns a record of all X'00'.

Wuerror

Wuerror indicates for CMS to return SFS extended error information.

REFRESH | NOREFRESH

REFRESH indicates that the most recent version of data requested from update in place files is retrieved. NOREFRESH means data is retrieved from CMS file buffers. REFRESH can be used in conjunction with the FORCE option of DMSWRITE to enable a reader or readers to see the updates made to an update in place file by a concurrent writer. REFRESH and NOREFRESH have no meaning for minidisk files.

requestid

Requestid is used to identify an asynchronous request. If it contains a binary 1 on input, then the request is to be asynchronous, and CMS generates an integer to identify the asynchronous request. If it is omitted or contains a binary 0 on input, the request is to be synchronous. This integer is placed in *requestid* which is passed on a later Check (DMSCHECK) request. All minidisk requests are done synchronously.

Writing Files

The DMSWRITE routine (Write) writes one or more records to a file. The file must have already been opened using DMSOPEN with the NEW, REPLACE, or WRITE parameter.

For a new file, writing begins with record one. For existing files, writing begins with the first record following the last record in the file. In both cases, you can indicate the number of the record to be written by using the *position* parameter.

In addition to the *retcode* and *reascode* parameters, specify the following parameters:

- *token*
- *records*
- *datalength*
- *buffer*.

The following parameters are optional:

- *position*
- *wuerror*
- FORCE | NOFORCE
- *requestid*.

Token

Token, returned from DMSOPEN, identifies the file to which you are writing.

Records

Records specifies the number of records to be written. This must be 1 for a file with variable-length records.

Datalength

Datalength specifies the maximum number of bytes to be written.

For fixed-length records, *datalength* must be specified as the product of *records* and the logical record length (lrecl):

datalength=records x *lrecl*

For variable-length records, it is the length of the record to be written:

datalength = 1 x *lrecl*

Variable-length records can be up to 65,535 bytes long. If you update an existing file of variable-length records, the replacement record must be the same length as the original record. If it is not the same length, then the write will fail and the file will remain as it was before the write.

Buffer

Buffer is the area containing the data to be written. *Buffer* must be large enough to handle the largest record to be written.

Position

Position specifies the record number of the first record to be written relative to the beginning of the file (record 1). The next sequential record is written if *position* is not specified or if 0 is specified (0 is the default).

If a file has fixed-length records, you can write a record beyond the current last record, including a record with a position number more than one greater than the number of the last record. If the skipped records are not written before the file is closed, the file is termed a sparse file. A skipped record may be written when the file is subsequently reopened. If a skipped record is read, it is retrieved as allX'00' bytes.

Wuerror

Wuerror indicates for CMS to return SFS extended error information.

FORCE | NOFORCE

FORCE causes updates of nonrecoverable files to be transmitted to SFS immediately. (FORCE is ignored for recoverable files.) NOFORCE lets CMS, not the application, control the transfer of data. FORCE can be used in conjunction with the REFRESH option of DMSREAD to enable a reader or readers to see the

updates made to an update in place file by a concurrent writer. FORCE and NOFORCE have no meaning to minidisk files.

requestid

Requestid is used to identify an asynchronous request. If it contains a binary 1 on input, then the request is to be asynchronous, and CMS generates an integer to identify the asynchronous request. If it is omitted or contains a binary 0 on input, the request is to be synchronous. This integer is placed in *requestid* which is passed on a later Check (DMSCHECK) request. All minidisk requests are done synchronously.

Altering Record Pointers

Another way to change the current read and write pointers in an open file is to use the DMSPOINT routine. This allows an application to change the read and write pointers for the next DMSREAD and DMSWRITE operations.

In addition to the *retcode* and *reascode* parameters, specify the following parameters:

- *token*
- *read_offset*
- *write_offset*
- *method*
- *new_read_pointer*
- *new_write_pointer*

Token

Token, returned from DMSOPEN, identifies the file to which you are changing.

Read Offset

Read Offset specifies where you want to move the read pointer (or if it is to be moved at all). If the value is zero, the pointer is not moved. See *Method* for more information.

Write Offset

Write Offset specifies where you want to move the write pointer (or if it is to be moved at all). If the value is zero, the pointer is not moved. See *Method* for more information.

Method

Method specifies how the read and write offsets should be applied to the current record pointers. The values for *method* are:

- 0 means that the offsets are absolute record pointers. The read and/or write pointers will be moved to these values. The exceptions are if the offset is a zero, then the pointer will not be changed. If the offset is -1, then the pointer is moved to the end of the file.
- 1 means that the values in *read_offset* and/or *write_offset* will be added to the current record pointers.
- 2 means that the values in *read_offset* and/or *write_offset* will be subtracted from the end of the file. This allows an application to move the record pointer based on the end of the file.

New Read Pointer

New Read Pointer provides the resulting read pointer when DMSPOINT completes.

New Write Pointer

New Write Pointer provides the resulting write pointer when DMSPOINT completes.

The various uses of *method* allow applications to adjust the read and write pointers anywhere in the file. Some examples of this are:

- To move only the write pointer to the end of file (past the last record):
  - Set the *write_offset* to -1
  - Set the *read_offset* to 0
  - Set the *method*to 0
- To move the read pointer ahead 10 records and the write pointer back 5 records:
  - Set the *write_offset* to -5
  - Set the *read_offset* to 10
  - Set the *method* to 1
- To move the read pointer 2 records from the end of the file and the write pointer 20 records past the end of the file:
  - Set the *write_offset* to -20
  - Set the *read_offset* to 2
  - Set the *method* to 2

Example: SFS Reading and Writing Records Sequentially

The following example shows how to use DMSREAD and DMSWRITE to read a record from one file and write it into another within the same work unit. The CACHE parameter on the DMSOPEN routine indicates that the data is read and written sequentially. Since it is within the same work unit, you do not have to specify the *workunitid*. Even though all of the examples in the following sections are SFS examples, changing the *dirname* to a file mode letter of an accessed minidisk would allow these programs to work on minidisk files.

```
/* REXX example of SFS read/write */
retcode=0
reascode=0
fileid1.fname = 'PROFILE'
fileid1.ftype = 'GIVE'
fileid1.dirname = 'SERVER8:FAIRLIEA.'
fileid1 = fileid1.fname fileid1.ftype fileid1.dirname
fileid1len = length(fileid1)
opentype.1 = 'READ'
opentype.2 = 'CACHE'
opentype = opentype.1 opentype.2
opentypelen = length(opentype)

call csl 'DMSOPEN retcode reascode fileid1 fileid1len opentype',
         'opentypelen tokenr'

records=1
datalen=80
buffer=0
bufferlen=80
bytesread=80

call csl 'DMSREAD retcode reascode tokenr records datalen buffer',
         'bufferlen bytesread'
⋮
fileid2.fname = 'PROFILE'
fileid2.ftype = 'GET'
fileid2.dirname = 'SERVER8:FAIRLIEA.'
fileid2 = fileid2.fname fileid2.ftype fileid2.dirname
fileid2len = length(fileid2)
opentype.1 = 'NEW'
opentype.2 = 'CACHE'
opentype = opentype.1 opentype.2
opentypelen = length(opentype)

call csl 'DMSOPEN retcode reascode fileid2 fileid2len opentype',
         'opentypelen tokenw'

call csl 'DMSWRITE retcode reascode tokenw records datalen buffer',
         'bufferlen'
⋮
exit
```

Example: SFS Reading and Writing of Variable-Length Records

When you read or write variable-length records, you must read one record at a time. The read/write buffer should be large enough to accommodate the largest record you read or write. Variable-length records can be up to 65,535 bytes long; if the record is longer than the buffer length, it is truncated. You cannot use null variable-length records.

When you read variable-length records, the *datalength* parameter should be equal to the maximum logical record length to avoid truncating the record, or

datalength=*lrecl* x 1

Buffer contains the data that was read and *bytesread* contains the number of bytes actually read. *Bytesread* contains 0 if the read failed with a return code of 8 or 12.

When you write variable-length records to a new file, writing begins with record one by default. When you write variable-length records to an existing file, writing begins with the first record following the end of the file by default. When you update a file 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 a return code of 8 and a reason code of 90121.

The following example shows how you could read and write a variable-length file. The "V" value, specified by *opentype.3* indicates that variable-length data is read or written.

```
/*                                     */
retcode=0
reascode=0
fileid1.fname = 'GIVEFILE'
fileid1.ftype = 'EXEC'
fileid1.dirname = 'SERVER8:FAIRLIEA.'
fileid1 = fileid1.fname fileid1.ftype fileid1.dirname
fileid1len = length(fileid1)
opentype.1 = 'READ'
opentype.2 = 'CACHE'
opentype.3 = 'V'
opentype = opentype.1 opentype.2 opentype.3
opentypelen = length(opentype)

call csl 'DMSOPEN retcode reascode fileid1 fileid1len opentype',
         'opentypelen tokenr'

records=1
datalen=130
buffer=0
bufferlen=130
bytesread=16

call csl 'DMSREAD retcode reascode tokenr records datalen buffer',
         'bufferlen bytesread'
⋮
fileid2.fname = 'GETFILE'
fileid2.ftype = 'EXEC'
fileid2.dirname = 'SERVER8:FAIRLIEA.'
fileid2 = fileid2.fname fileid2.ftype fileid2.dirname
fileid2len = length(fileid2)
opentype.1 = 'NEW'
opentype.2 = 'CACHE'
opentype.3 = 'V'
opentype = opentype.1 opentype.2 opentype.3
opentypelen = length(opentype)

call csl 'DMSOPEN retcode reascode fileid2 fileid2len opentype',
         'opentypelen tokenw'

call csl 'DMSWRITE retcode reascode tokenw records datalen buffer',
         'bufferlen'
⋮
exit
```

Example: SFS Reading and Writing of Specific Records

CMS keeps pointers to keep track of which records were last written and read. To read or write a specific record, you can specify the *position* parameter of the DMSREAD or DMSWRITE routines.

To read a specific record declare the *position* parameter as a signed binary integer and initialize it as the value of the record number that you want to read. If the record is beyond the end of the file, no data is placed in the buffer and a return code of 4 is returned to warn you that the end of the file was reached.

To write a specific record, declare the *position* parameter as a signed binary integer and initialize it as the value of the record number that you want to write. You can write a record beyond the current last record in the file by specifying a record number that is one or more than the number of the last record. If you do not write the skipped records before you close the file, the file is considered a sparse file. You can write the skipped records when the file is subsequently reopened. Reading a skipped record places all X'00' bytes in the buffer.

The following example shows how you could read record 5 in ONE FILE and then write it as record 8 in TWO FILE. The NOCACHE option indicates that the data is read and written randomly (not sequentially).

```
/*                    */
retcode=0
reascode=0
fileid1.fname = 'ONE'
fileid1.ftype = 'FILE'
fileid1.dirname = 'SERVER8:FAIRLIEA.'
fileid1 = fileid1.fname fileid1.ftype fileid1.dirname
fileid1len = length(fileid1)
opentype.1 = 'READ'
opentype.2 = 'NOCACHE'
opentype.3 = 'F'
opentype = opentype.1 opentype.2 opentype.3
opentypelen = length(opentype)

call csl 'DMSOPEN retcode reascode fileid1 fileid1len opentype',
         'opentypelen tokenr'

records=1
datalen=80
buffer=0
bufferlen=80
bytesread=5
position=5

call csl 'DMSREAD retcode reascode tokenr records datalen buffer',
         'bufferlen bytesread position'
⋮
fileid2.fname = 'TWO'
fileid2.ftype = 'FILE'
fileid2.dirname = 'SERVER8:FAIRLIEA.'
fileid2 = fileid2.fname fileid2.ftype fileid2.dirname
fileid2len = length(fileid1)
opentype.1 = 'WRITE'
opentype.2 = 'NOCACHE'
opentype.3 = 'F'
opentype = opentype.1 opentype.2 opentype.3
opentypelen = length(opentype)
position = 8

call csl 'DMSOPEN retcode reascode fileid2 fileid2len opentype',
         'opentypelen tokenw'

call csl 'DMSWRITE retcode reascode tokenw records datalen buffer',
         'bufferlen position'
⋮
exit
```

Example: SFS Reading and Writing of Specific Records Using DMSPOINT

This example is identical to the previous example except that it uses DMSPOINT to manipulate the record pointers.

```
/*                    */
retcode=0
reascode=0
fileid1.fname = 'ONE'
fileid1.ftype = 'FILE'
fileid1.dirname = 'SERVER8:FAIRLIEA.'
fileid1 = fileid1.fname fileid1.ftype fileid1.dirname
fileid1len = length(fileid1)
opentype.1 = 'READ'
```

```
opentype.2 = 'NOCACHE'
opentype.3 = 'F'
opentype = opentype.1 opentype.2 opentype.3
opentypelen = length(opentype)

call csl 'DMSOPEN retcode reascode fileid1 fileid1len opentype',
         'opentypelen tokenr'

writeoffset=0
readoffset=5
method=0
newwritepointer=0
newreadpointer=0

call csl 'DMSPOINT retcode reascode tokenr writeoffset readoffset',
         'method newwritepointer newreadpointer'
records=1
datalen=80
buffer=0
bufferlen=80
bytesread=5
position=0

call csl 'DMSREAD retcode reascode tokenr records datalen buffer',
         'bufferlen bytesread position'
.
.
.
fileid2.fname = 'TWO'
fileid2.ftype = 'FILE'
fileid2.dirname = 'SERVER8:FAIRLIEA.'
fileid2 = fileid2.fname fileid2.ftype fileid2.dirname
fileid2len = length(fileid1)
opentype.1 = 'WRITE'
opentype.2 = 'NOCACHE'
opentype.3 = 'F'
opentype = opentype.1 opentype.2 opentype.3
opentypelen = length(opentype)
position = 0

call csl 'DMSOPEN retcode reascode fileid2 fileid2len opentype',
         'opentypelen tokenw'

writeoffset=8
readoffset=0
method=0
newwritepointer=0
newreadpointer=0

call csl 'DMSPOINT retcode reascode tokenw writeoffset readoffset',
         'method newwritepointer newreadpointer'

call csl 'DMSWRITE retcode reascode tokenw records datalen buffer',
         'bufferlen position'
.
.
.
exit
```

Closing Files

It is important to close files you are working with as soon as you are finished with them. This minimizes the amount of user virtual machine resources required and helps keep shared files and directories available for other users. Closing a file means logically disconnecting it from the application program.

Note that closing a file in an SFS directory control directory does not always make the file available to other users for writing. If you have a directory control directory accessed in read/write status, no other user can write to any file in the directory so long as you have the directory accessed. Closing the file, in this case, does not make the file available to others for writing.

The DMSCLOSE routine (Close) closes files previously opened by using the DMSOPEN routine. You pass the token from DMSOPEN to identify the files that you are closing. You may also want to include the COMMIT parameter to commit the changes made to the files after they are closed, if you want to ensure any uncommitted changes.

Example: To close and commit an output file that has been updated, issue:

```
commit='COMMIT'
comlen=length(commit)
```

```
call csl 'DMSCLOSE retcode reascode tokenw commit comlen'
```

Truncating Files

The CSL routine, DMSTRUNC, allows programs to delete records from the end of a file. DMSTRUNC returns a file that has the number of records that was specified by the program on input. Truncation can only be done on a closed file. For example, if the file TOOBIG FORME in the POOLA:SCOTT.TEST directory has 1000 records and you want to truncate it to 500 records, you can specify the following REXX statements:

```
fileid.fname = 'TOOBIG'
fileid.ftype = 'FORME'
fileid.dirname = 'POOLA:SCOTT.TEST.'
fileid = fileid.fname fileid.ftype fileid.dirname
fileidlen = length(fileid)
numrecs = 500
commit='COMMIT'
comlen=length(commit)

call csl 'DMSTRUNC retcode reascode fileid fileidlen numrecs commit comlen'

exit
```

You can also specify an optional date and time parameter on the call to DMSTRUNC. This gives the program the ability to set the file's update date and update time to something different than the current system date and time.

You can also produce an empty SFS file using DMSTRUNC. This can be accomplished by setting the number of input records to zero and by specifying ALLOWEMPTY. If ALLOWEMPTY is not specified or the file is on a minidisk, this will fail with an error return code and reason code.

Erasing Files

Use the DMSERASE routine to delete a minidisk file, Shared File System base file, alias, external object, or directory; or to delete the data in an SFS file. For example, to delete the file GETRID OFITNOW in the POOLA:JOHN.TEST directory, use the following REXX statements:

```
fileid.fname = 'GETRID'
fileid.ftype = 'OFITNOW'
fileid.dirname = 'POOLA:JOHN.TEST.'
fileid = fileid.fname fileid.ftype fileid.dirname
fileidlen = length(fileid)
options='COMMIT ENTIRE'
optlen=length(options)

call csl 'DMSERASE retcode reascode fileid fileidlen options',
         'optlen'

exit
```

The authorities that you have to a file and a directory affect whether you can use the ENTIRE and DATAONLY options to erase another user's files and aliases. This table summarizes the interactions for both files and aliases:

Table 12. Using the DMSERASE routine with the ENTIRE and DATAONLY options

Call DMSERASE against	Option	Necessary authorities			Results
		File write	Directory write	File read	
File in file control directory	ENTIRE	•	•		File and all related authorizations, aliases, and control data are erased.
File in file control directory	DATAONLY	•			Only contents of file are deleted; aliases, authorizations, control data, and empty file remain.

Table 12. Using the DMSERASE routine with the ENTIRE and DATAONLY options (continued)

Call DMSERASE against	Option	Necessary authorities			Results
		File write	Directory write	File read	
Alias	ENTIRE		•	•	Alias is deleted; base file is unaffected.
Alias	DATAONLY	•			Contents of base file are deleted.
File in directory control directory	ENTIRE		•		File and all related authorizations, aliases, and control data are erased.
File in directory control directory	DATAONLY		•		Only contents of file are deleted; aliases, authorizations, control data, and empty file remain.

You cannot erase a file under the following conditions:

- You have the file open
- The file is locked by another user
- You are not authorized to the file or directory
- Any user (including the issuer of your program) has a SHARE lock on the file.

If the issuer of your program has an UPDATE or EXCLUSIVE lock on a file, the file can be erased.

For more information on aliases and sharing files, see "SFS File Sharing" on page 172. For more information on locks for files and directories, see "Locking SFS Files and Directories" on page 176.

Committing Your Changes

When you complete your file SFS I/O, you need to tell CMS what to do with the changes you made to the file. The changes can be saved, or *committed*, or they can be discarded, or *rolled back*. There are several ways to commit changes made in programs:

- Supply a COMMIT parameter in the parameter list of the DMSCLOSE.
- Call the DMSCOMM (Commit) CSL routine.
- Call the SRRCMIT (Commit) SAA resource recovery routine.
- Execute a commit verb of a resource participating in CRR.

The preferable way of committing changes is to supply a COMMIT parameter on the DMSCLOSE routine. However, if you issue the DMSCOMM and DMSCLOSE routines individually and receive any error messages, you can determine which routine generated the error.

To roll back the changes to a recoverable file, execute a SRRBACK (Backout) SAA resource recovery routine, DMSROLLB (Rollback) CSL routine, or a rollback verb of a resource participating in CRR.

Changes to minidisk files can not be explicitly committed or rolled back. Changes are committed when the last file on the minidisk that is opened for output is closed. Using the COMMIT and NOCOMMIT parameters (as well as DMSCOMM and DMSROLLB) apply only to changes made to SFS and other resources participating in CRR for that work unit. For more information on committing and rolling back changes, see "Committing and Rolling Back Changes in Application Programs" on page 141. See Chapter 16, "Your Applications and Data Integrity," on page 241 for details of how CRR coordinates commits and rollbacks for protected resources (those that participate in CRR).

Data Block I/O

The File I/O routines that have been previously discussed do all I/O to CMS files (minidisk and SFS) at the record level. That is, all reading and writing to CMS files are done with record lengths and number of records in mind. A file record is the lowest unit of file I/O that can be performed to a CMS file. While this

is adequate in most cases, it can be inefficient to use this interface to transfer large files. For example, DMSREAD allows an application to read 1 or more records from a file. If the file has a fixed record format, the number of records that can be read is determined by the size of the application's storage buffer. In the case of a variable record format file, only one record can be read at a time. This can cause significant overhead for the system to read or write a large file.

The data block interface can be used to provide a high performance general interface for data transfer to and from CMS files (both minidisk and SFS). This is not intended for applications that wish to modify existing CMS files. Applications wishing to modify CMS files should use the DMSOPEN/DMSREAD/DMSWRITE/DMSCLOSE routines.

Using this interface, an application can read and write data blocks to CMS files. For fixed format files, a data block contains only the file data. In variable format files, a data block contains records made up of a two byte length prefix which contains the length of the record, then the data itself. The CMS file system uses these record lengths in variable format files to know where and how long the next record is. If an application modifies these record lengths incorrectly, it may cause unpredictable results when the file is read later by the file system.

There are four CSL routines that are used for data block I/O:

- DMSOPDBK - Open Data Block
- DMSRDDBK - Read Data Block
- DMSWRDBK - Write Data Block
- DMSCLDBK - Close Data Block

The data block I/O routines work in much the same manner as the corresponding record I/O routines. When a file has been opened using the data block I/O interface, you must use the corresponding data block I/O routines to read, write and close the file. You could not, for example, use DMSREAD to read a block when the file was opened using DMSOPDBK.

One primary use of the data block I/O interface is to transfer CMS files. This can be done between:

- Minidisks
- Minidisks and SFS file pools
- SFS file pools

When the file is opened using DMSOPDBK, the block size of the file is returned. For SFS files, this is always 4096 (4KB). For minidisk files, this can be 512, 1KB, 2KB or 4KB. The application can use the Read Data Block (DMSRDDBK) routine to read a source file into a buffer. The application can open a target file for output (New or Replace) and use the Write Data Block (DMSWRDBK) routine to write the data from the buffer used by DMSRDDBK. The buffer being written must be as least as large as the block size of the output file. If it is larger, the file system will correctly adjust the blocks for the target file. As you can see, if you use data buffers in multiples of 4KB, all blocking will be handled by the file system and does not have to be handled by the application.

One significant difference between the record interface and data block interface is what happens when the file is closed. Using the record interface the file system knows how many records have been written. When the file is closed, the number of records written and the appropriate record size is saved as attributes of the file. When using the data block interface, the file system only knows the number of blocks that have been written. It isn't until the file is closed (using DMSCLDBK), that the application has to provide the number of records and record length of the file. The file system uses this to compute where the application wants the end of the file to be. This may not agree with the number of data blocks that have been written to the file.

In the case of a variable format file, the file system determines where the last record of the file ends. This is indicated when a record length prefix of zero is found. If this does not agree with the number of records that was given by the application to DMSCLDBK, a warning return code and the correct number of records and record length is returned.

In the case of a fixed format file, the file system uses the input to DMSCLDBK to determine where the actual end of the file is. If the number of records and record length from the application is less than the

number of blocks written, the file will be truncated to that size. If the applications indicates the end of the file is past where the last data block has been written, the file will be extended with sparse blocks (data blocks containing binary zeroes). Another difference between the data block and record interfaces is when SFS data is committed. Using the record interface, you can commit changes to the file while the file is open. For the data block interface, the file system knows less about the file so the file must be closed before the changes can be committed. In fact, there can not be any file open using DMSOPDBK on the work unit in order to commit the changes to the work unit.

You can operate on files simultaneously using both FS macros and data block I/O routines. The following describes how each method can be used for both minidisk and SFS files.

- When using DMSOPDBK:
  - For an SFS file, you can open the file more than once for input (read) AND only once for output (new or replace).
  - For a minidisk file, you can open the file more than once for input (read) OR once for output (new 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.

Directory I/O

This section describes how to use CMS routines to manage SFS and minidisk directories. The following program functions are provided for SFS directory manipulation:

Table 13. Program Functions for SFS Directory Manipulation

CSL Call	Function	Description
DMSCLDIR	Close Directory	Logically disconnects an application program from a specific SFS directory.
DMSCRDIR	Create Directory	Creates a new directory
DMSDIRAT	Set Directory Attribute	Sets or changes the directory attribute.
DMSERASE	Erase	Erases directories (files, and aliases) and external objects.
DMSEXIDI	Exist - Directory	Checks for an existing directory and return the directory information in variables.
DMSEXIST	Exist	Checks for an existing directory (or file) and return the directory information in a buffer.
DMSGETDA	Get Directory - Searchall	Reads one or more directory records into variables when a directory has been opened with an intent of SEARCHALL.
DMSGETDD	Get Directory - Dir	Reads one or more directory records into variables when a directory has been opened with an intent of DIR.
DMSGETDF	Get Directory - File	Reads one or more directory records into variables when a directory has been opened with an intent of FILE.
DMSGETDI	Get Directory	Reads one or more directory records into a buffer.
DMSGETDK	Get Directory - Lock	Reads one or more directory records into variables when a directory has been opened with an intent of LOCK.
DMSGETDL	Get Directory - Alias	Reads one or more directory records into variables when a directory has been opened with an intent of ALIAS.

Table 13. Program Functions for SFS Directory Manipulation (continued)

CSL Call	Function	Description
DMSGETDS	Get Directory - Searchauth	Reads one or more directory records into variables when a directory has been opened with an intent of SEARCHAUTH.
DMSGETDT	Get Directory - Auth	Reads one or more directory records into variables when a directory has been opened with an intent of AUTH.
DMSOPDIR	Open Directory	Logically connects an application program to a specific SFS directory for subsequent reading.
DMSRELOC	Relocate	Moves a subdirectory (or file) from one directory to another.
DMSRENAM	Rename	Renames a subdirectory (or file).

Here is a list of some of the tasks you can perform on SFS directories:

- Determine if the SFS directory exists — You can use the DMSEXIST or DMSEXIDI routines to determine if the SFS directory you want to work with already exists or whether you need to create a new one. You may want to skip this step and just open the directory; the status of the directory could change from the time you determine its existence to the time you actually open it.
- Open a directory to read it — Use the DMSOPDIR routine to logically connect your program to a specific directory. You can open the directory more than once.
- Read directory records — The Get Directory routines allow you to scan files and subdirectories by reading records in the directory (previously opened using DMSOPDIR). You can also scan for lock, authority, and alias information.
- Create or erase an SFS directory — The DMSCRDIR allows you to create a new directory. DMSERASE erases directories.
- Close a directory — The DMSCLDIR routine logically disconnects your application program from a specific directory (previously opened using DMSOPDIR).
- Commit your changes — Use the DMSCOMM routine to commit changes that you have made to the work unit, or use the COMMIT parameter on the DMSCLDIR routine.

Minidisks also have a directory, known as the "Master File Directory" that holds information about CMS files. This directory can be scanned to find information about the files on the minidisk. The following are the CSL routines that you can use to reference minidisk directories.

Table 14. CSL Routines for Manipulating Minidisk Directories

CSL Call	Function	Description
DMSCLDIR	Close Directory	Logically disconnects an application program from a specific directory.
DMSGETDF	Get Directory - File	Reads one or more directory records into variables when a directory has been opened with an intent of FILE.
DMSGETDI	Get Directory	Reads one or more directory records into a buffer.
DMSOPDIR	Open Directory	Logically connects an application program to a specific directory for subsequent reading (open intent of FILE only).

Here is a list of some of the tasks you can perform on minidisk directories:

- Open a directory to read it — Use the DMSOPDIR routine to logically connect your program to a specific minidisk directory. The FILE intent is the only intent that can be used with a minidisk directory.
- Read directory records — The Get Directory routines allow you to scan files by reading records in the directory (previously opened using DMSOPDIR). You can also use DMSGETDF (Get Directory for FILE) to retrieve this information

- Close a directory — The DMSCLDIR routine logically disconnects your application program from a specific directory (previously opened using DMSOPDIR).

Determining If an SFS Directory Exists

To determine if a directory exists in the Shared File System, and the status of the directory, use the DMSEXIDI or DMSEXIST routine. However; if you plan to do subsequent reads or writes to the directory **and** share the directory with other users, you should not bother with determining if the directory exists. You should begin directory I/O by opening the directory. This assures that no other users could erase or revoke the directory before you get a chance to open the directory. Once you have the directory opened, other users can not erase the directory.

DMSEXIDI returns directory information in variables. If the directory does not exist or you do not have authority on the directory, DMSEXIDI returns a return code of 8 as the value of the return code parameter and a reason code of 44000 as the value of the reason code parameter. If the directory does exist, DMSEXIDI returns a return code of 0 in the return code parameter and places the directory information in the variables that you provide.

DMSEXIST returns directory information in a specified buffer. If the directory does not exist or you do not have authority on the directory, DMSEXIST returns a return code of 8 as the value of the return code parameter and a reason code of 90230 as the value of the reason code parameter. If the directory does exist, DMSEXIST returns a return code of 0 in the return code parameter and places the directory information in the specified buffer.

You can map the output of DMSEXIST with the EXSBUFF assembler mapping macro. For more information on EXSBUFF, see the *z/VM: CMS Macros and Functions Reference*.

Opening and Reading SFS Directories

Once you know that a directory exists, you can open it; that is, you can establish a logical connection to the directory for subsequent reading. When you open a directory, CMS passes a token back to your program. The token is an 8-byte field that you pass on to other routines for reading the directory entries and closing the directory. A minidisk directory may be opened and read in the same manner as an SFS directory. This only applies to the Open Directory intent of FILE. All other Open Directory intents are for SFS directories only.

Opening Directories

The DMSOPDIR routine lets you open a directory and specify the intent, or the type of read that you will perform. The parameter that you specify determines the type of information that will be available on subsequent Get Directory routines.

Specify one of the following parameters to indicate the intent:

- FILE
- FILEEXT
- SEARCHALL
- SEARCHAUTH
- ALIAS
- AUTH
- LOCK
- DIR.

Position is set to the top of the directory. If a directory is open for anything except FILE, you may not call any atomic requests on that work unit for that file pool until the work unit is committed. (For more information on atomic requests, see page “Atomic Requests” on page 139.)

FILE

Specify the file name and file type (or namedef) parameter with the directory name, file mode letter (or namedef) and FILE; you must have READ authority, WRITE authority, or both on the SFS directory to use FILE. A minidisk must be accessed in order to issue the directory functions. Only the FILE intent may be used to scan a minidisk directory. Information returned in later DMSGETDI and DMSGETDF calls consists of:

- File identifier
- File mode letter
- File mode number
- Record format
- Logical record length
- Number of blocks
- Number of records
- Date of last update
- Time of last update
- User ID of the owner
- Type (base, alias, erased, revoked, directory, or external object or minidisk file)
- Authority
- Directory attribute
- Unique ID
- Migration indicator.

DMSGETDI also returns the type of open (has a value of 1 for FILE).

Opening a directory with intent FILE causes CMS to access the SFS directory implicitly. Regular ACCESS command default rules are used to determine whether the access is read-only or read/write, unless the FORCERO or FORCERW options are specified when DMSOPDIR is done. Note that a read-only access of directory control directory prevents you from writing to that directory or to the files it contains (even if you are an administrator). If you need to write, access the directory control directory in read/write status before you execute DMSOPDIR with intent FILE.

FILEEXT

Specify the file name and file type (or namedef) parameter and FILEEXT. Information returned from a later DMSGETDI and DMSGETDX consists of everything that is returned when FILE is specified, plus the:

- File space type (0 = SFS)
- Date of last reference
- Creation date
- Creation time
- Recoverability attribute
- Overwrite attribute
- Number of data blocks used for the file
- Number of system blocks used for the file
- Maximum blocks used for the file
- DFSMS/VM related attributes

DMSGETDI also returns the type of open (has a value of 8 for FILEEXT).

SEARCHALL and SEARCHAUTH

Specify the file name and file type (or namedef) parameter with the directory name, file mode, or namedef and SEARCHALL or SEARCHAUTH. The directory and all associated subdirectories are searched. SEARCHAUTH only searches and returns files for which the issuer is authorized. Information returned in later DMSGETDI, DMSGETDA, and DMSGETDS calls consists of:

- File identifier
- File mode number
- Record format
- Logical record length
- Number of blocks
- Number of records
- Date of last update
- Time of last update
- User ID of the owner
- Type (base, alias, erased, revoked, directory, or external object)
- Authority (only for SEARCHAUTH)
- External protection (only for SEARCHAUTH)
- Directory name and length
- Directory attribute
- Migration indicator.

DMSGETDI also returns the type of open (has a value of 2 for SEARCHALL and 3 for SEARCHAUTH).

ALIAS

Specify the file name and file type (or namedef) parameter with the directory name, file mode, or namedef and ALIAS. You must have read authority on the file name specified. If special characters (* and %) are specified to designate a set of files, you must have read authority on the directory. Information returned from later DMSGETDI and DMSGETDL calls consists of:

- Input file identifier
- Input file mode number
- Type (base, alias, erased, revoked)
- Number of aliases
- Directory name and length
- Output file identifier
- Output file mode number
- User ID of the owner
- Migration indicator.

If the input file on the open directory is a base file, then the file's aliases are returned. If the input file on the open directory is an alias, then the file's base file information is returned. DMSGETDI also returns the type of open (has a value of 4 for ALIAS).

AUTH

Specify the directory name, file mode, or namedef and AUTH. The file name and file type (or namedef) parameter is optional. When you use AUTH, later DMSGETDI calls only returns information about authorizations that the issuer has and those that the issuer has granted.

You must have read authority on the file name specified. If special characters (* and %) are specified to designate a set of files, you must have read authority on the directory.

Information returned in later DMSGETDI and DMSGETDT calls consists of:

- File identifier (can be a file name and file type or a subdirectory)
- File mode number
- Type (base, alias, erased, revoked, or directory)
- Read authority
- Write authority
- External protection
- User ID of the owner
- Directory attribute.
- Migration indicator.

DMSGETDI also returns the type of open (has a value of 5 for AUTH).

LOCK

Specify the directory name, file mode, or namedef and LOCK. The file name and file type (or namedef) parameter is optional. You must have read authority on the file name specified. If special characters (* and %) are specified, information returned from later DMSGETDI and DMSGETDK calls consists of:

- File space type (0 = SFS)
- File identifier (can be a file name and file type or a subdirectory)
- File mode number
- Type (base, alias, erased, revoked, or directory)
- Lock type (share, exclusive, or update)
- Lock length (session or lasting)
- Lock user ID.
- Migration indicator.

DMSGETDI also returns the type of open (has a value of 6 for LOCK).

DIR

Specify the directory name, file mode, or namedef and DIR. The file name and file type (or namedef) must not be specified. Each subdirectory is returned on later GET DIRECTORY calls. The directory name consists of the file pool ID, user ID, and up to 8 subdirectory names. Information returned from later DMSGETDI and DMSGETDD calls consists of:

- Directory length
- Directory name
- Directory attribute.

DMSGETDI also returns the type of open (has a value of 7 for DIR).

For the exact output information for each of the intent parameters, see the DMSGETDI routine (Get Directory) in the *z/VM: CMS Callable Services Reference*.

Reading Directories

Once the directory is open, use the Get Directory routines to read one or more records. The first Get Directory call reads the first record; subsequent calls read the next records. Remember to specify the token from the DMSOPDIR routine to uniquely identify the open directory.

When reading records, you can choose between two methods: you can have the entries returned into a buffer or they can be returned in variables.

Use DMSGETDI to read the record into a buffer. The type and length of directory record(s) read into the buffer depend on the type of intent specified on DMSOPDIR. All records are fixed-length. Fields that are not used are set to blanks (if a character field) or zero (if a numeric field). Use the DIRBUFF assembler macro to map the output. For more information on DIRBUFF, see the *z/VM: CMS Macros and Functions Reference*.

If, due to a language restriction, you cannot or do not want to provide a structure to handle the data returned in a buffer, you can use Get Directory routines that return the available information in variables. The routine that you use corresponds to the type of intent specified on DMSOPDIR.

Use	For DMSOPDIR with an intent of
DMSGETDF	FILE
DMSGETDX	FILEEXT
DMSGETDA	SEARCHALL
DMSGETDS	SEARCHAUTH
DMSGETDL	ALIAS
DMSGETDT	AUTH
DMSGETDK	LOCK
DMSGETDD	DIR

No implicit SFS locks are held across the Open Directory (DMSOPDIR), Get Directory (DMSGETDD, DMSGETDF, DMSGETDI, and so on), and Close Directory (DMSCLDIR) sequence. This means that while a directory is open, if the issuer or another user changes information in the directory, the change may or may not be reflected in subsequent Get Directory requests.

When a directory is open, for an intent other than FILE, 16KB of data is sent to the user's virtual machine for use by Get Directory. If changes are made while the directory is open, (for example, a file is erased) those changes are not reflected in the data in the user machine. However, if Get Directory needs to get another block of data from the server, the changes may be reflected in the new set of data.

An exception to this rule is when the directory is open for intent FILE. In this case, changes may be immediately reflected and subsequent Get Directory requests will reflect those changes. For example, if the directory is opened for intent FILE and a file in the directory is erased, subsequent Get Directory requests will not return an entry for that file.

An exception to this exception is when the intent is FILE and the work unit it used was gotten for another user ID (see DMSGETWU in the *z/VM: CMS Callable Services Reference* for more details.) In this case, changes are not immediately reflected and subsequent Get Directory requests work in the same manner as for intents other than FILE.

Closing Directories

Use the DMSCLDIR routine (Close Directory) to close a directory that has been opened using DMSOPDIR (Open Directory). Closing a directory logically disconnects your program from a specific directory so that you can no longer read records in it. You identify the directory by passing the token from the DMSOPDIR routine (Open Directory).

Creating a Directory in SFS

You can create a directory or subdirectory in a file pool using the DMSCRDIR routine (Create Directory). Remember the following rules when creating directories:

- You cannot create a top directory using DMSCRDIR. A top directory is automatically created when you are enrolled in a SFS file pool.
- Including the top directory, nine levels of directories are the maximum that can be created.
- You can only create a directory structure one level at a time.

- You cannot create a directory in another user's directory structure. However, the SFS administrator can create a directory in another user's directory structure.

You can commit the newly created directory by including the commit parameter in the routine.

Erasing a Directory in SFS

The DMSERASE routine (Erase) erases base files, aliases, directories, and external objects. Remember the following rules when erasing directories:

- Only the directory owner or the SFS administrator can erase a directory.
- To erase a directory that contains files, you need to use the FILES parameter on the DMSERASE routine and to be sure that all the files are closed. If there are files in the directory, they are erased also. If the directory does not contain any files, you can omit the FILES parameter.
- A directory cannot be erased if:
  - The directory is open as a result of a DMSOPDIR call. However, if the directory is open with an intent of FILE, but the FILE and NOCOMMIT parameters are not specified together on the erase request, then the directory, even though open, will be erased. You can also erase files in a directory that is open.
  - The directory is locked by another user.
  - Any user (including the issuer of DMSERASE) has a SHARE lock on the directory. If the issuer of DMSERASE has an UPDATE or EXCLUSIVE lock on the directory, the directory **can** be erased.
  - The directory contains subdirectories.

For information on erasing directories, see the DMSERASE routine in the *z/VM: CMS Callable Services Reference*.

Committing Your Changes

When you complete your directory I/O, you need to tell the SFS file pool server what to do with the changes you made to the directory on the work unit. The changes can be saved, or *committed*, or they can be discarded, or *rolled back*. There are several ways to commit changes made in programs:

- Supply a COMMIT parameter in the parameter list of the DMSCLDIR, DMSCRDIR, or DMSERASE routine.
- Call the DMSCOMM (Commit) CSL routine.
- Call the SRRCMIT (Commit) SAA resource recovery routine.
- Execute a commit verb of a resource participating in CRR.

The preferred way of committing changes is to supply a COMMIT parameter on the DMSCLDIR, DMSCRDIR, or DMSERASE routine. However, if you issue the DMSCOMM and DMSCLDIR, DMSCRDIR, or DMSERASE routines individually and receive any error messages, you can determine which routine generated the error.

To roll back changes made to the directory, execute a SRRBACK (Backout) SAA resource recovery routine, DMSROLLB (Rollback) CSL routine, or a rollback verb of a protected resource. For more information on committing and rolling back changes, see “Committing and Rolling Back Changes in Application Programs” on page 141. See Chapter 16, “Your Applications and Data Integrity,” on page 241 for details of how CRR coordinates commits and rollbacks for protected resources (those that participate in CRR).

Changes to minidisks are done when the last file opened for output on a file mode is closed. Minidisk operations are not associated with work units. The COMMIT and ROLLBACK functions affect the specified work unit, and have no effect on the minidisk operations that have been completed.

SFS File Sharing

When writing application programs using SFS, you can share files and directories in several ways. However, remember, file sharing is invoked by the owner authorizing other users to read or modify a file or directory.

Files may be shared using three different mechanisms:

- Create aliases to files
- Access directories (using the ACCESS command)
- Reference files directly using the File I/O routines in VMLIB.

Following are some of the File I/O routines that you will use when sharing files and directories.

Table 15. SFS Routines for Sharing Files and Directories

CSL Call	Function	Description
DMSGRANT	Grant Authority	Permits another user to use a file or directory.
DMSCRALI	Create Alias	Places an additional name for a file in a directory.
DMSERASE	Erase	Erases aliases, files, directories, and external objects.
DMSREVOK	Revoke Authority	Revokes use of a file or directory.

Granting Authority for Files and Directories

To share files using any of the three methods just mentioned, you first need to grant authority to other users to use a file or directory. Use the DMSGRANT routine (Grant Authority) to grant authority to other users.

When granting authority in application programs, keep the following rules in mind to avoid confusion about authorizations:

- The issuer must own the file or directory to grant authority on it. The SFS administrator can also grant authority on any file or directory.
- If the issuer grants authority on an alias, the authority refers to the base file.
- An authority on a file in a file control directory does not imply that you have any authority on the directory in which the file resides.
- Authorizations cannot be granted on individual files within directory control directories or on aliases that refer to base files in directory control directories.
- READ or WRITE authority on a file control directory does not imply that you have any authority on any file within the directory. For example, READ authority on a file control directory allows another user to see the names of a file in the directory but not to read a file.
- Authority can be granted to a file in a file control directory even if the file is open or if the file or directory is locked, except when the file is locked exclusively by someone else.

To specify the name of the file or directory, use the actual identifiers or use namedefs. If you specify a file name or namedef for a file, you must also specify the directory in which it resides. If you omit the file name and file type or namedef for the file, then the name or namedef specified is assumed to be the directory to which authority is being granted.

Use the READ, WRITE, NEWREAD, NEWWRITE, DIRREAD, DIRWRITE, and READDIRREAD parameters in DMSGRANT to specify the type of authority being granted. For files in file control directories, you can grant READ and WRITE authority. For file control directories, you can grant READ, WRITE, NEWREAD, or NEWWRITE authority. For directory control directories you can grant DIRREAD and DIRWRITE authority. You cannot grant authority to individual files in directory control directories. For more information about these authorities, see the DMSGRANT routine description in *z/VM: CMS Callable Services Reference*.

All users must be authorized to communicate with an SFS file pool server. This may be done on an individual user basis or by specifying that anyone may communicate with it.

The file pool administrator controls authorization to an SFS file pool server. The file pool administrator allows anyone to communicate with an SFS server by issuing the ENROLL PUBLIC command. The file pool administrator allows an individual to communicate with an SFS server through the ENROLL USER

command. ENROLL USER gives an individual a top directory. ENROLL PUBLIC and ENROLL USER allow a user to read or modify any file or directory for which he has those authorizations.

Additionally, the user may be authorized to use a specified amount of logical space in a particular SFS file pool or may not be allocated any space. The file pool administrator also does this through the ENROLL USER command on which the amount of space authorized can be specified. The space usage limit may be modified through the MODIFY USER command. This user may then create directories and files, and may authorize other users to use those files.

An alternative to using the DMSGRANT or DMSREVOK routines, is to use an external security program to determine who may read or modify your files.

An external security program may coexist with SFS base authorizations. This means that if the program does not know about the object, SFS authorizations are used. Optionally, an external security program may be brought up to respond "unauthorized" if it does not know about the object.

Creating Aliases to Files

One way of sharing files is by creating an ***alias*** to a file. An alias provides a means of getting to information in a base file without moving any data or without creating a copy of the file. The alias is simply a pointer to the base file. It may be in the same directory as the base file or in a different one. You can also create an alias on an alias.

Aliases cannot be created in directory control directories, on minidisks or to minidisk files. They must reside in file control directories. Aliases can, however, refer to files that reside in directory control directories. You cannot create an alias on a file that is open for write.

Use the DMSCRALI routine (Create Alias) to create an alias. You can use namedefs to specify the:

- Base file
- Base file directory
- Alias
- Directory that will contain an alias.

To have your program create an alias:

- The issuer must be the owner of the base file or must have been granted the proper authorization. For base files in file control directories, READ or WRITE authority to the file is required. For base files in directory control directories, DIRREAD or DIRWRITE authority on the directory in which the base file resides is required.
- The issuer must have WRITE authority on the file control directory where the alias is being created.
- The directory owner of the directory that will contain the alias must have authority to the base file.

When creating aliases, they may not cross SFS file pools.

For more information on creating aliases, see the *z/VM: CMS Callable Services Reference* and the *z/VM: CMS User's Guide*.

External Objects

While aliases cannot cross file pools, ***external objects*** can refer to objects in different file pools. An external object is another type of entity managed by SFS, in addition to files and aliases to files. An external object is an SFS directory entry containing the name (called the ***remote name***) of some object. This object may be a file, but does not have to be. Like a file or alias, an external object has a unique name, appears on a FILELIST screen, and can be renamed, relocated and erased (but not locked). Unlike a file or alias, however, the actual data referenced by the remote name in an external object is not managed by your local SFS server. You cannot directly manipulate it. It may reside in another file pool, minidisk or data base. SFS external object support is an application-enabling facility. External objects are intended for CMS applications which have requirements for mapping SFS file IDs to objects outside of the file pool. The meaning of the information stored in an SFS external object is entirely application-defined.

You can create an external object using the DMSCROB routine. You can query the remote name contained in an external object using the DMSQOBJ routine. For more information on DMSCROB and DMSQOBJ, see the *z/VM: CMS Callable Services Reference*.

Accessing Directories

Another way to share files is by using the ACCESS command. Using this method, you would grant authority for a directory to another user, such as USERA. That user would then use the ACCESS command to use the directories and the files in it.

Once the directory is accessed, USERA can issue commands on that directory. By default, the CMS File System treats access to another user's directory as an access in read-only mode. (Otherwise, programs using STATEW would assume that they could create a new file on that directory.)

When you access directory control directory in read-only status, you cannot write to anything in the directory, even if you own the directory or have DIRWRITE authority to it. Commands, CSL routines, or macros that try to write to the directory will fail.

When you access a file control directory in read-only status, however, you can write to files in the directory if you are properly authorized. To write to the files, you can use CSL routines or a few CMS commands. Most CMS commands and assembler language macros require the directory to be accessed in read/write status. XEDIT and COPYFILE commands do not. You can use XEDIT to edit another user’s file in a directory that you have accessed read-only. You can also use COPYFILE. In either case, you must have WRITE authority to the file in order to write to it. To have XEDIT and COPYFILE respect the read-only access, use the SET RORESPECT ON command.

If you want to execute commands that require a read/write file mode, you can force another user's directory to be accessed in read/write status. To do so, specify the FORCERW option on the ACCESS command.

You can force a file control directory into read/write status even if you have only READ authority to the directory. Naturally, if you try to do something (such as creating a file) that requires WRITE authority, the operation will fail. You cannot circumvent SFS authority checking by forcing a read/write access.

When forcing a read/write access to a another user's file control directory, keep in mind that some programs may have compatibility problems. A program might, for example, assume that it can create new files when, in fact, the user has only READ authority on the directory.

Unlike file control directories, you cannot force a directory control directory into read/write status without being authorized to write. You must have DIRWRITE authority to the directory control directory or your attempt to force it into read/write status will fail. Because of this authorization requirement, forcing another user's directory control directory into read/write status does not cause program compatibility problems. Programs can safely assume that they can create or erase files in the directory, and that they can write to any file in the directory.

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

Direct File Reference

The third method of sharing a file is direct file reference through the direct file reference routines for SFS in VMLIB. Your program can request a specific file in a specific directory which need not be accessed. However, note that the user running the program must be authorized to use that file.

You can also add a file to another user's directory if you are properly authorized. For file control directories, you must have WRITE authority to the directory. For directory control directories, you must have DIRWRITE authority. When you create a file in another user's file control directory, you have WRITE authority to the file, but the directory owner owns the file. The directory owner can revoke your WRITE authority.

When you create a file in another user's directory control directory, you can write to the file by virtue of your DIRWRITE authority on the directory in which the file resides. But, if the owner of the directory decides to revoke DIRWRITE authority, you lose your ability to write to the file.

By using direct reference you can avoid having your application access directories. With file control directories, it does not matter whether the directory is or is not accessed. If it is accessed, it does not matter if it is accessed read-only or read/write.

With directory control directories, however, it does matter. To successfully write to a directory control directory using direct reference, you must either not have the directory accessed at all, or you must have the directory accessed in read/write status. A read-only access will prevent you from writing to the directory.

Using direct reference to write to a file in a directory control directory will also fail if another user has the directory accessed in read/write status. Once anyone has a directory control directory accessed in read/write status, no other user can write to it.

Sharing Files and Directories

If your program creates files or directories or both, consider the following when determining how to share them:

- Sharing through an alias is most appropriate where:
  - There are multiple users authorized to write.
  - The people sharing wish to have different names for the files or to group them differently.
- Sharing by the ACCESS command is most appropriate where:
  - A group of people wish to share an entire directory (using the ACCESS command)
  - A user wants to edit a file (through XEDIT) but does not want a permanent alias.
- Sharing through direct file reference is appropriate for:
  - An application written to use the CSL routines.

Removing Authority for Shared Files and Directories

Use the DMSREVOK routine (Revoke Authority) to remove authority that has been granted using DMSGRANT. You can also change a use DMSREVOK to downgrade a user's authority (from WRITE to READ, for example). Remember the following rules when using DMSREVOK to remove or change authorities within a program:

- The issuer must be the SFS administrator or the owner of the file or directory to revoke or change authority.
- The issuer cannot revoke authority from himself.
- Authority can be revoked from a file that is open; it takes effect when the file is closed.
- Authority cannot be revoked from a directory if there are files open within that directory.
- Authority cannot be revoked on a file or directory if another user has locked the file or directory.
- Authority cannot be revoked on a file or directory if the issuer has locked the file or directory using SHARE.

Locking SFS Files and Directories

This section describes locking in SFS and how to use CMS routines to lock SFS files and directories. The following routines are provided for file and directory locking:

CSL Call	Function	Description
DMSCRLOC	Create Lock	Creates an explicit lock on a file or directory.
DMSDELOC	Delete Lock	Deletes an explicit lock on a file or directory.

Locking

When more than one person can write to a file or directory, there must be some way to ensure that two people do not update the same object at the same time. The SFS locking mechanism provides as much concurrency as possible. To prevent simultaneous updates, SFS uses a locking scheme composed of implicit and explicit locks.

An ***implicit lock*** is one that SFS acquires and releases automatically. Whenever someone has a file open, SFS internally associates an implicit lock with that object. If someone else tries to read or write to that file, SFS first checks whether it is locked before allowing access. An ***explicit lock*** is one that you create by issuing a CMS command or routine that forces an object to be locked. Both kinds of locks are discussed in the next two sections.

Locks have two important characteristics: duration and type. ***Lock duration***, simply, is the length of time the lock exists. The ***lock type*** determines whether others can share the object during the time it is locked.

Implicit Locking

Implicit locks are automatically acquired by the server to ensure data integrity among multiple users. To allow greatest concurrency, SFS acquires locks only when needed and frees them as soon as it can. The lock duration is the length of a server unit of work, which is, for example, from the time a file is opened until it is closed and any changes committed. Note that if you commit the changes, but have not yet closed the file, the implicit locks are still held. Similarly, if you update a file and then close it, but do not commit the work unit, the locks are still held. Maintaining the implicit lock eliminates the need to:

- Use multiple work units or explicit locks to guarantee a consistent version of an update-in-place input file
- Reopen directories
- Reopen input files (and possibly see a later version of the data).

Internally, SFS uses several different lock types. For our purposes, however, it is best to think of implicit locks as having only two types: share and exclusive.

SHARE
: An implicit share lock permits multiple readers of an object. SFS acquires an implicit share lock when a user opens an object for read. When an object is implicitly share locked, other users can implicitly lock the object as share or exclusive.

EXCLUSIVE
: An implicit exclusive lock permits only one writer of an object. SFS acquires an implicit exclusive lock when a user opens an object for anything other than read. When an object is implicitly exclusive locked, other users can implicitly lock the object as share, but they cannot implicitly lock the object as exclusive.

The lock type and duration for implicit locks depends on the operation you ask SFS to perform. SFS, for instance, gets different locks for an open for write or replace requests (DMSOPEN routine with the WRITE or REPLACE parameter) and for an erase request (DMSERASE) even though these operations are considered a type of write. Some locks can be freed when the operation completes, while other locks must be held until the work unit in which the operation occurs is committed. In any case, all implicit locks acquired during a work unit are freed when the work unit is rolled back. Explicit locks, which we will be discussing next, can last beyond the work unit (even if there is a rollback).

When you try to update an object that is implicitly exclusive locked, your request is rejected. If you made the request in a program, your program will receive a return code of 8. If you made the request by issuing a CMS command, the command will fail.

Explicit Locking

When you let SFS lock and unlock your files automatically, the implicit lock will remain in effect until the file is closed and any changes have been committed or rolled back. If you want a lock to remain in effect longer than this, you must explicitly lock the object by entering a CREATE LOCK command before running your program or by issuing a DMSCRLOC routine (Create Lock) within your program. This section discusses

the DMSCRLOC routine. For more information on the CREATE LOCK command, see the *z/VM: CMS User's Guide* and the *z/VM: CMS Commands and Utilities Reference*.

The DMSCRLOC routine allows you to explicitly lock a file or a directory. Explicit locks are sometimes called *check-out* locks because you are checking-out a file or directory just as you would check out a book at a library.

With the DMSCRLOC routine you can specify both the duration of the lock and the type of lock.

You can specify three types of explicit locks:

SHARE
An explicit share lock allows multiple readers, but no writers. Explicit share locks can be created on file control directories and the files within. Explicit share locks cannot be created on directory control directories or files in those directories. Multiple users may have a share lock on the same file or file control directory at the same time.

If you create a share lock you will not be able to write to the file or directory even if you have WRITE authority. But, you will be able to read it, as will others.

READ authority is required for the specified file or directory.

EXCLUSIVE
An explicit exclusive lock type means there can be only one person accessing the file or directory at a given time. Explicit exclusive locks can be created on file control directories and the files within. Explicit exclusive locks cannot be created on directory control directories or files in those directories. If you create an exclusive lock, you are the only person that can read or write the file or directory. It also prevents other users from getting any locks on the file or directory.

To obtain an exclusive lock, there cannot be any other types of locks on the file or directory.

WRITE authority is required for the specified file or directory.

UPDATE
An explicit update lock type allows others to read while you read or update.

In the case of file control directories, the only person allowed to write to the file or directory is the issuer of the CREATE LOCK command or DMSCRLOC routine. In the case of directory control directories:

- A lock on a **file** prevents everyone **except the holder** of the lock from accessing the directory read/write.
- A lock on a **directory** prevents everyone **including the holder** of the lock from accessing the directory read/write.

No other user is allowed to update the file or directory even if the issuer is not currently updating the file or directory.

For file control directories and files within them, WRITE authority is needed to create an update lock. For directory control directories and files within them, DIRWRITE authority on the directory is needed to create an update lock.

The **duration** of the lock can be:

SESSION
An explicit session lock lasts until the end of the CMS session, or until it is specifically deleted (with the DELETE LOCK command or DMSDELOC routine), or until all user machine connections to the SFS file pool are broken (for example, with the DMSPURWU routine).

LASTING
An explicit lasting lock lasts until it is deleted with the DELETE LOCK command or DMSDELOC routine. The lock lasts across CMS sessions and logon sessions. If, for example, you create a LASTING lock and log off, the object as still locked, even though you are not logged on.

You should use explicit locks whenever you want to control the activity on your files or directories without revoking authority. If, for example, you are rewriting a document that everyone has access to, and you do not want anyone to see the new draft until it is complete, you might create lasting exclusive locks

on the document's files. If, on the other hand, you do not care whether anyone sees the changes, but want to make certain no one else is changing the files, you would create lasting update locks. Once the document is complete you would either delete the locks or create lasting share locks if you wanted to prevent changes.

To request a lock, all activity in the affected SFS file pool for the work unit must be committed. If there is any outstanding work on the file pool, the request fails.

Relationships between Locks

Table 16 on page 179 depicts the relationship between the various locks. Remember the following rules when considering locks:

- Implicit locks are acquired by SFS.
- Explicit locks are acquired by users (or by some CMS commands like XEDIT or SET LANGUAGE).
- A explicit share lock has *n* readers and no writer.
- An explicit exclusive lock has no readers and one writer.
- An update lock has *n* readers and one writer.
- To obtain any type of lock, there cannot be any other types of locks on the file or directory.

Only UPDATE locks are allowed on directory control directories and on the files within them. So, cases involving share and exclusive locks in the figures are not applicable to directory control directories and the files within them.

Table 16. Results of Interactions between Accessing and Locking

		When you try to				
		Read	**Write**	**Create a Share Lock**	**Create an Exclusive Lock**	**Create an Update Lock**
If someone is	reading	OK	OK	OK	fail/wait	OK
	writing	OK	fail/wait	fail/wait	fail/wait	fail/wait
If someone has already created	a share lock	OK	fail	OK	fail	fail
	an exclusive lock	fail	fail	fail	fail	fail
	an update lock	OK	fail	fail	fail	fail

Legend:

OK
Done immediately.

fail
Command fails or program receives an error code.

fail/wait
Indicates action with FILEWAIT OFF/ON, which applies only to implicit locks. See *z/VM: CMS Commands and Utilities Reference* for information on the SET FILEWAIT command.

Table 17 on page 180, Table 18 on page 180, Table 19 on page 180, and Table 20 on page 180 show what happens when you try to create a lock on a file or directory using DMSCRLOC when the object is already locked.

Table 17. Issuer Has Lock on File and Requests Another on Directory

Lock Mode for Directory	Lock Mode for File		
	Explicit Share	Explicit Update	Explicit Exclusive
Explicit Share	Yes	No	No
Explicit Update	Yes	Yes	Yes
Explicit Exclusive	Yes	Yes	Yes

Table 18. Issuer Has Lock on Directory and Requests Another on File

Lock Mode for File	Lock Mode for Directory		
	Explicit Share	Explicit Update	Explicit Exclusive
Explicit Share	Yes	Yes	Yes
Explicit Update	No	Yes	Yes
Explicit Exclusive	No	Yes	Yes

Table 19. Issuer Requests Lock on Directory While Another User Has Lock on File

Lock Mode for Directory	Lock Mode for File		
	Explicit Share	Explicit Update	Explicit Exclusive
Explicit Share	Yes	No	No
Explicit Update	Yes	No	No
Explicit Exclusive	No	No	No

Table 20. Issuer Requests Lock on File While Another User Has Lock on Directory

Lock Mode for File	Lock Mode for Directory		
	Explicit Share	Explicit Update	Explicit Exclusive
Explicit Share	Yes	Yes	No
Explicit Update	No	No	No
Explicit Exclusive	No	No	No

Deleting Locks

The DMSDELOC routine (Delete Lock) releases an explicit lock on a file or directory that was created with the CREATE LOCK command or DMSCRLOC routine. Only the creator of the lock or an SFS file pool administrator can delete it.

To delete a lock, all activity in the affected SFS file pool for the work unit must be committed. If there is any outstanding activity, the request fails. If you want to delete a lock in an active SFS file pool, you can use another work unit. For example, say there is an open file in SFS file pool A, which is active in the default work unit, and you want to delete the lock on that file. You can call DMSGETWU to get another work unit ID and then call DMSDELOC (on the new work unit ID) to delete the lock in SFS file pool A.

Waiting for Locks

If your request fails because the object is locked, you should wait until the object is not locked and reissue the request. Programs that fail because of lock conflicts should be rerun when the objects it needs are not locked.

If you do not care how long you might wait for a response to a request, and if you are being rejected because you are the second writer, you can tell SFS to wait for the object to become unlocked rather than reject the request. To do this, you issue a SET FILEWAIT ON command. However, this only can help if the files or directories are locked implicitly.

This is useful, for example, at the end of the day, when you might start a program, disconnect your virtual machine, and then go home. It is not a good idea to submit a job to the batch machine that issues a SET FILEWAIT ON command. While your program is waiting, so are all the other programs in the batch machine queue.

Because most CMS commands issued from a terminal form a single logical unit of work, the implicit lock is usually not held for a long time. In application programs, however, you have control over the committing of work units and can cause implicit locks to be held for a long time.

Deadlocks

In any system that manages shared resources, it is possible for deadlocks to occur. A ***deadlock*** is a standstill that is reached when two or more users are each waiting for a resource that the other holds.

In SFS, it is possible for deadlocks to occur only for implicit locks. Because SFS never waits for an explicitly locked object, even if FILEWAIT is ON, there cannot be a deadlock that involves any previously explicitly locked object—SFS would have already terminated the request.

A deadlock would occur if USERA's program holds a lock on FILEA while waiting for a lock on FILEB. Meanwhile, USERB holds a lock on FILEB while waiting for a lock on FILEA. All the locks are implicit locks, obtained, perhaps, by opening a file for write. USERA is waiting for USERB, while USERB is waiting for USERA. If nothing was done, both USERA and USERB would wait forever.

Or, suppose user USERA's program holds an implicit lock on FILEA while waiting for an explicit lock on FILEB. Meanwhile, user USERB holds an implicit lock on FILEB while waiting for an explicit lock on FILEA, and both USERA and USERB have SET FILEWAIT ON. USERA is waiting for USERB, while USERB is waiting for USERA. This also would cause a deadlock to occur. If nothing was done, both USERA and USERB would wait forever.

SFS detects these situations when the deadlock is contained within a single file pool, and SFS resolves them by rolling back the youngest logical unit of work (the one that started most recently). SFS will roll back the logical unit of work even if that user had entered SET FILEWAIT ON.

If you forget what locks you have created on your files or directories, issue the QUERY LOCK command. The QUERY LOCK command also displays the locks created on your file or directory by other users to whom you have granted read or write authority. You can also use the DMSOPDIR (Open Directory) routine with DMSGETDI (Get Directory) or DMSGETDK (Get Directory - Lock).

Who Is Locking the SFS File or Directory?

Some CMS commands do not have specific return codes that indicate the command failed because the file was locked. Instead, a return code indicating an *open* error is returned. While this allows existing programs to run without having to be recoded and recompiled, it does prevent a CMS user from knowing immediately whether the command failed because there was truly a file open error or because the file was locked.

If commands operating on files or directories repeatedly fail, even though you know the object exists and you are authorized to access it, you should suspect that the object is locked. The message or error code that describes the failure will indicate either an error opening the object or a locking error.

There are two ways to check whether a file or directory is locked. One method checks for explicit locks, which most frequently cause lock conflicts. The other method checks for implicit locks. Because implicit locks are usually held for a short time, they are not often the cause of repeated command failure. By the time you reissue the command, the implicit lock is often freed, and the command succeeds. So, the first kind of lock you should look for is an explicit lock.

Checking Explicit (or Check-Out) Locks

To check whether a file or directory is explicitly locked, issue the QUERY LOCK command. You must be authorized for the file or directory that you query. For example, suppose user CROCKETT, a travel writer, has a file space in the SFS file pool named TRAVEL. Crockett wants to see whether the .MAPS.USA directory is locked by one of his co-workers. He would issue:

```
query lock travel:crockett.maps.usa
```

Or, if the default SFS file pool for Crockett is TRAVEL, he could use the abbreviated form:

```
query lock .maps.usa
```

If Crockett accessed the directory as, for example, file mode B, he can also issue:

```
query lock b
```

If the directory is not locked, CMS will display a response saying so. Otherwise, Crockett would see a display like this:

```
Directory = TRAVEL:CROCKETT.MAPS.USA
User ID    Lock      Duration
TWAIN      EXCLUSIVE LASTING
```

The display shows the directory, TRAVEL:CROCKET.MAPS.USA, exclusively locked by Twain. "Lasting" means that the directory is locked regardless of whether Twain is logged on.

To check whether the file NEWYORK SCRIPT is explicitly locked within that directory, Crockett would issue:

```
query lock newyork script .maps.usa
```

or

```
query lock newyork script b
```

Again Crockett would see either a message telling him the file was not locked, or a display like this:

```
Directory = TRAVEL:CROCKETT.MAPS.USA
Filename Filetype Fm   Type  User ID    Lock      Duration
NEWYORK  SCRIPT   B1   BASE  JIM        SHARE     SESSION
NEWYORK  SCRIPT   B1   BASE  TOM        SHARE     LASTING
```

Here the type is BASE, which means that NEWYORK SCRIPT is a base file. Two users, Jim and Tom have share locks on the file. Jim's lasts until the end of his session, while Tom's lasts until he deletes it. The B1 file mode indicates that Crockett still has the .MAPS.USA directory accessed as B.

For application programs, you can use the DMSOPDIR (Open Directory) routine with DMSGETDI (Get Directory) or DMSGETDK (Get Directory - Lock). Specify DMSOPDIR with the intent of LOCK. Subsequent DMSGETDI or DMSGETDK calls provide the lock type, lock length, and lock user ID

If you suspect that a file or directory you are trying to use is locked, and QUERY LOCK doesn't show it as being locked, check for an implicit lock.

Checking Implicit Locks

Because implicit locks are fleeting, the QUERY LOCK command does not display information about them. If it did, there would be a good chance that the result would not be valid by the time the result was displayed.

To check for implicit locks:

1. Enter SET FILEWAIT ON.

   This command tells CMS to wait for the file to become free if a lock conflict occurs. By default, FILEWAIT is OFF, so CMS will not wait and the command fails.

2. Reissue the command that was causing the lock problem.

   If the object is still implicitly locked, the command will wait. Go to the next step.

   If the command succeeds immediately, the implicit lock must have been recently freed. In this case, just issue SET FILEWAIT OFF, and continue your work.

   If the command fails, someone must have just acquired an explicit lock, or the object no longer exists, or you are no longer authorized to access the object.

3. Have a friend issue a QUERY FILEPOOL CONFLICT command for you. If, for instance, Crockett was trying to see whether an implicit lock was causing him problems in the TRAVEL SFS file pool, he would have a friend issue:

   ```
   query filepool conflict crockett travel:
   ```

   The result will look like this:

   ```
   Requester   Holder     Wait             Lock          Lock Type
   ---------   --------   --------------   -------       -------
   TWAIN       SMITH      Lock             File          Excl
   LEIGH       SMITH      Lock             File          Excl
   SUE         SMITH      Lock             File          Excl
   CROCKETT    SMITH      Lock             File          Excl
   ```

   The first two columns show the most important information: who is requesting the lock and who is holding the lock. (We are going to ignore the other columns for now.) Crockett, along with three other users, are waiting for Smith. The other users are listed because they are in queue for the resource ahead of Crockett, which means that if Smith suddenly freed his lock, Crockett would still have to wait for Twain, Leigh, and Sue.

   Note: QUERY FILEPOOL CONFLICT does not show conflicts caused by explicit locks because CMS never waits for an explicit lock, even if FILEWAIT is on.

Once you find out who is holding the lock, you can call them or send them a message to see when they will be done. If you choose not to wait, and you want to cancel the command, re-IPL CMS.

When you IPL CMS, SFS realizes that you have ended your CMS session, and stops waiting to process your command. During CMS initialization, FILEWAIT is automatically reset to OFF.

Canceling a Command When FILEWAIT Is On

When FILEWAIT is ON, it is possible that your command or program will take a long time to complete. Suppose you change your mind about waiting for the command or program. How do you cancel it? You can:

- Enter the #CP LOGOFF command.
- Enter #CP IPL CMS. (You can IPL a system other than CMS.)
- Have your SFS file pool server operator issue a server FORCE operator command to sever your links to the SFS file pool server.

Any of the actions causes the communications between your virtual machine and any other SFS file pool server machine to be severed. SFS file pool server machines detect this and immediately stop processing your command. There is, however, a remote chance that while you are logging off or issuing the IPL command, the SFS file pool server machine can complete processing your original request. In that case, you may not see the successful completion message. So, in your next CMS session it is a good idea to look at files that would have been changed to see whether they were changed.

Note that in XEDIT each subcommand is a complete operation. That is, at the completion of an XEDIT subcommand, any file read from or written to is left closed. So, any XEDIT subcommand that reads from or writes to a file, such as LOAD, PUT, or FILE, may be delayed if FILEWAIT is ON. If FILEWAIT is OFF and someone else is using a file, then a LOAD, PUT, or FILE would fail for that file.

Performance Tips

When writing application programs that use files in SFS or minidisks, several factors can influence how well your program performs. The following sections list the performance considerations.

SFS Performance Tips

The following are performance considerations for your programs when using SFS files:

- If your application will perform only a few operations to a file or directory, then use direct reference to files and directories to avoid processing for the ACCESS command.
- If your application will perform many operations on files, then accessing the directories (instead of direct reference) may make the operations process more efficiently.
- Use hierarchical directories to minimize the number of files that need to be accessed. Only access those files that are actually needed during normal application processing.
- If your application deals with read-only or read-mostly data, consider using directory control directories. If you are using an S/390® processor, your administrator can make directory control directories eligible for use in a data space. This can improve the read performance of your application. See the *z/VM: CMS User's Guide* for more about the use of directory control directories.
- Refer to a file by its base file name rather than through one of its aliases.
- Rather than issuing a separate request, such as STATE or DMSEXIST, to see if a file exists before opening it, you should attempt to open the file and check the return codes to determine if it exists. Not only is this more efficient, but it also eliminates the possibility that the status of the file has changed between the time the existence check was made and the time the file was opened.
- If the directory containing a file has been accessed, it is normally more efficient to use DMSEXIFI, rather than DMSEXIST, to check for the existence of a file or to obtain file-related information. DMSEXIST always results in a file pool server call. If the directory is accessed, DMSEXIFI does not result in a file pool server call (unless the file is protected by an external security manager or variables following *wuerror* are specified in the parameter list). Instead, file status is obtained directly from the cached directory information in the user virtual machine. SFS ensures that this data is kept up to date.
- It is good practice to close files and commit your work as soon as possible, as this will minimize real storage requirements and the probability that other users will experience a lock conflict with files that your application is using. In particular, in an interactive application, try to avoid being in a unit of work while the user is in think time. Issue prompts before opening files and directories.
- Reduce the number of file system calls. For example, if you wish to close a file and commit your work, issue DMSCLOSE with the commit option rather than use separate DMSCLOSE and DMSCOMM calls.
- In cases where the application wishes to make an exact copy of an existing file and both files are in the same SFS file pool, it is more efficient to use the DMSFILEC routine to do this than for the application to read and rewrite the file. DMSFILEC will do this with one file system call.
- When you wish to move a file from one directory to another within the same file space, consider using DMSRELOC. This is more efficient than copying the file and then erasing the original file because no file data is actually moved, no file data has to be erased, and all authorizations and aliases associated with the file are retained.
- When replacing the contents of a file, make use of the REPLACE option to directly replace the file rather than creating a temporary file, erasing the original file, and renaming the temporary file. SFS ensures that the original file will remain intact until you commit your work. If you use the REPLACE option to replace the contents of a file, you will not lose any of the aliases or authorizations that were set.
- If one or more input files is in migrated status, use the DFSMS RECALL command to bring their contents into the SFS server prior to running the application. This may be done asynchronously.
- You may also improve performance for migrated files by issuing DMSOPEN requests asynchronously for input files.

SFS and Minidisk Performance Tips

The following are performance considerations for your programs when using minidisk or SFS files:

- Sequential processing of fixed-length records can be made more efficient if the application provides a large buffer and reads/writes multiple logical records in one request. For optimal performance, the buffer should start on a page boundary.
- When opening a file, it is best to explicitly specify CACHE or NOCACHE, as this ensures that the file system will select the most efficient mode of processing.
- For applications coded in assembler, you can reduce the processing required to call repeatedly executed CSL routines by use of the CSLFPI macro. See "Invoking CSL Routines Frequently from Assembler Programs" on page 326 for further information.

Using SFS File Space

Remember that before you create an SFS file in a file pool, you must be enrolled in that file pool and assigned an allocation of space. This allocation of logical space is called a file space. The owner of the file space, the SFS administrator, and any user authorized by the owner can create files in that file space.

Directories are in a separate file space area, an area of system owned space. Therefore, your directory entries do not take up your own file space. The file pool catalog, within the system owned space, contains information about the files and directories that exist in the file pool, such as, who owns them and who is authorized to look at them.

Threshold Warning

When you are enrolled in an SFS file pool, a threshold warning point is set on space usage. When this point is reached, a warning will be reported in one of the following ways:

- If you are using an SFS routine, your program receives a return code of 4 and a reason code of 51050.
- If you are using a CMS command, a warning message will be issued.
- If you are using the FS macros (for example, FSWRITE), the FSCBTHEX bit will be set in the File System Control Block (FSCB).

Be aware that this threshold does not prevent you from continuing to use additional space. Until a COMMIT operation is attempted, a threshold warning may be the only indication that your application will get, even if the file space limit has been exceeded.

A default threshold percentage of 90% is set when you are enrolled with file space in a file pool, but you can modify your own file space threshold percentage with the SET THRESHOLD command. File space usage and threshold percentage may be queried with the QUERY LIMITS command or DMSQLIMU routine.

Temporary Space

SFS file space and minidisk space are handled differently in CMS. For example, if you have a program that writes intermediate files on a user's minidisk, you have to be sure there is enough minidisk space to hold the file or obtain temporary minidisk space. In SFS, the file space allocated to each user is considered committed space. Because you can use space above your allocated space until you commit, your program can write files and close them, but not commit the files. You can then reopen the file and read the uncommitted updates, and continue processing.

If your application is written using SFS routines, and it is unable to commit a work unit because file space limits have been exceeded, the commit failure will not automatically cause activity on the work unit to be either committed or rolled back. Your application must either delete unneeded file blocks that are above the file space limit and attempt to commit the work again, or explicitly roll back the changes if there is not sufficient space to hold them.

If your application does not use CSL routines to update SFS files, an error will be returned for a write operation when the SFS file space limit is detected. In an environment where files are not shared, you should have enough space to close and commit all updated files. In an environment where there is

sharing, updates on the work unit will be rolled back if the SFS file space limit is exceeded when the files are closed and changes are committed.

In either case, you could lose some of the changes you intended to be permanent if temporary changes and permanent changes are on the same logical unit of work. Keep in mind that this may happen even if the file is nonrecoverable.

Your installation may have established a storage policy and installed a file space usage exit that may prevent or limit overuse of uncommitted space. Contact your file pool administrator for details.

Accessing Multiple SFS File Pools

In CMS, you can update multiple SFS file pools and other protected resources on a single work unit. Whenever a commit (or rollback) is issued on that work unit, CRR coordinates the commit (or rollback) among all of the resources to ensure data integrity.

See Chapter 16, “Your Applications and Data Integrity,” on page 241 for information on accessing multiple file pools.

SFS Restart Recovery

Not all sudden failures of your computer system cause a data loss. When an SFS server is started, it determines whether anyone was making changes when it last ended. That is, SFS tries to find unfinished work. If it does, it automatically rolls back any uncommitted changes. By doing this, SFS automatically recovers from almost all system failures.

If your application also accesses other protected resources, this same protection is extended to those other resources through Coordinated Resource Recovery (CRR). For details on CRR, see Chapter 16, “Your Applications and Data Integrity,” on page 241.

SFS User Synchronization

SFS synchronizes files to allow concurrent access to a file by multiple people. Multiple readers are allowed to a single file, along with one writer.

Writers can always see the changes they are making. That is, they can read records they have just written without closing and reopening the file.

Readers see changes at varying times, depending primarily on the file overwrite attribute.

Synchronization for NOTINPLACE Files

Readers can see only committed changes to NOTINPLACE files. (Most files have the NOTINPLACE attribute.) Once the writer commits the change, there may be a delay before a reader sees the updates.

For NOTINPLACE files in file control directories, users who do not already have the file open see the changes as soon as they open the file. If the file was open by the reader before the writer committed the changes, the reader must close and reopen the file to see the changes. Once the file is open, the reader has a consistent view of the file. That is, the version of the file the reader sees is frozen the instant the file is opened.

For NOTINPLACE files in directory control directories, different rules apply. In this case, the access status (or lack of it) affects when a reader sees the updates. If the reader does not have the directory control directory accessed, the rules are the same as those for file control files. That is, the reader must close and reopen the file to see the changes. If, however, the directory is accessed, the reader will not see the changes until the reader reaccesses the directory. This characteristic is sometimes referred to as ***access-to-release consistency*** and is similar to the consistency provided for minidisk files.

Synchronization for INPLACE Files

Readers can see uncommitted changes to SFS INPLACE files. Once the changes are written to the file pool, they are available to readers regardless of the directory attribute (FILECONTROL or DIRCONTROL) and the access status.

It is possible for readers not to see changed records because an old version of the record is already in their virtual machine's file buffers. Readers can use the REFRESH option on the DMSREAD routine if they want to be certain they are reading the latest record.

For INPLACE files, SFS does not try to ensure any sort of consistency. You can read a record twice and get different results if someone managed to write a record between your reads.

Another potential delay in reading changes is caused when files are extended. A file extension is any new block or record added to the file. To see extensions to INPLACE files in file control directories, the reader must close and reopen the file. To see extensions to INPLACE files in directory control directories, the reader must reaccess the directory if it was accessed. If it was not accessed, the reader must close and reopen the file to see the extensions.

Lock Collisions

Within the SFS file pool server, there are implicit waits. For example, if a user attempts to open a file for write which is already open for write by another user, and the SET FILEWAIT command was issued with ON, the SFS file pool server waits until the file is closed and committed. If SET FILEWAIT is OFF, the default, there is a rejection of requests instead of waiting on an implicit lock. No waiting ever occurs on a collision with an explicit lock. For example, if a user attempts to open a file for write that has an explicit share lock on it, the request is rejected, regardless of the FILEWAIT setting.

Asynchronous Requests

Most CSL routines that interface with the Shared File System (such as DMSOPEN, DMSERASE, DMSCRLOC) can be issued either synchronously or asynchronously by an application. To process a routine asynchronously means that the server handles the request while control returns to your program so it can continue processing. Asynchronous support is designed for performance-sensitive applications such as service machines or multitasking applications that may be doing other work concurrently.

Note: If you are writing an application that issues asynchronous SFS requests on behalf of other users, you may wish to consider using multiple user ID support.

You can indicate that a request is to be processed asynchronously by specifying a binary 1 in the *requestid* parameter. This causes CMS to generate an integer that uniquely identifies the asynchronous request. This integer is passed back in the *requestid* parameter of the routine making the request. You can then specify this value on DMSCHECK to determine if the request has completed. (If a 1 is passed back, it means that no server interaction was required to complete the request.)

Note: All requests to a minidisk will be processed synchronously. If an asynchronous request was issued (requestid = binary 1), the request is still processed synchronously to the minidisk and the requestid will still have a 1 when the CSL routine has completed.

CMS passes the asynchronous request on to the server for processing. When control is returned to your program, the return and reason codes indicate whether the request was accepted or immediately rejected. If the return code is zero, indicating that the request was accepted, your program needs to invoke the DMSCHECK routine with the value returned in *requestid* to check if the asynchronous request has completed.

You must specify a WAIT or NOWAIT parameter on the DMSCHECK routine. The WAIT parameter means that your program waits until the request you are checking for completes. The NOWAIT parameter means that control returns to your program to continue processing.

Note that a number of server requests may be needed to satisfy a single CSL request. This is particularly true when large amounts of data are being processed. When DMSCHECK is specified with the NOWAIT option, you may need to issue DMSCHECK more than once to complete the request.

The request is not considered to be complete until the DMSCHECK has completed. Both input and output parameters of the initial CSL request may be updated by CMS at any time during this process. The application must not change the contents of these parameters until DMSCHECK completes, or unpredictable results may occur. When DMSCHECK completes, you must examine the return code and reason code parameters of the initial request (for example, DMSOPEN) to determine the outcome of that operation.

If the return code on DMSCHECK is zero, the request has completed and the variables for the request are filled in. A return code of 4 indicates that the request has not yet completed. A return code of 8 indicates that an error occurred. See the *z/VM: CMS Callable Services Reference* for further information on DMSCHECK.

If there are any asynchronous requests pending when CMS end-of-command processing takes place, the work units with which these requests are associated will be rolled back.

Even when you request asynchronous processing, there are several circumstances in which some synchronous SFS server processing may be performed to satisfy your request. These circumstances include:

- You specify the COMMIT keyword and other CRR-participating resources (for example, another file pool) are active for this work unit.
- You specify the COMMIT keyword and you have open SFS output files on the work unit.
- Your asynchronous CSL request is the first SFS request to the file pool for the specified work unit.

Note: You cannot call any CSL routines asynchronously from a REXX program.

If a work unit has an active asynchronous request, you cannot issue any other request for the affected file pool on that work unit. The only exception to this rule is a rollback request, which could be a rollback routine, such as DMSROLLB, or a routine or macro that causes a rollback, such as the DMSPURWU (Purge Work Unit IDs) routine or the DMSABN macro.

Issuing Asynchronous SFS Requests from CMS Multitasking Applications

A multitasking application that issues asynchronous requests to an SFS server can use DMSCHECK to test for the completion of the requests:

- **Use CMS Application Multitasking Event Management routines with DMSCHECK**. Only the thread issuing the request ceases operation. This section describes how to use this technique.
- **Issue the DMSCHECK CSL routine with the NOWAIT option until the request has been completed**. Processing continues, but the application must keep issuing DMSCHECK until the request has been completed. This technique allows other threads in the application to continue, but they must compete for processor time with the repeated calls to DMSCHECK.
- **Issue the DMSCHECK CSL routine with the WAIT option**. The virtual machine goes into a wait state, and all threads of the application wait until the request being checked has been completed.

The first of these techniques blocks only a single thread while an asynchronous request is processed. It uses Event Management routines and a system event that is reserved for signaling the completion of asynchronous CSL routine requests to an SFS server. The characteristics of this event are:

Characteristic
Description

Name
VMSFSASYNC

Scope
Session. All processes in the session can monitor and signal this event.

Signal Delivery
CMS simultaneously signals the completion of the asynchronous request to all qualifying monitors.

Signaler
Asynchronous. CMS continues its internal processing after it signals the completion of the asynchronous request.

To implement this technique, the application must:

1. Call DMSGETWU to get a work unit ID.
2. Create an event key for each combination of file pool ID and work unit ID being used. Each event key is 12 bytes long and is composed of the 8-byte character file pool ID concatenated with the 4-byte integer work unit ID: FFFFFFFFWWWW.
3. Call the EventMonitorCreate function for the event name VMSFSASYNC and for the event keys of the requests (see note "1" on page 189, below).
4. Issue asynchronous requests to the SFS server. You can have only one outstanding asynchronous request per file pool ID-work unit ID pair.
5. Check that a request was not processed synchronously (see note "2" on page 189, below).
6. Call EventWait with the event token for the requests (see notes "1" on page 189 and "3" on page 189, below). Only the thread that calls EventWait is blocked. When CMS signals the completion of any SFS request that matches the specified event token, the thread resumes execution.
7. Call DMSCHECK to determine how the asynchronous request was completed. DMSCHECK can be issued at any time after EventWait and can be issued with either the WAIT or NOWAIT option (see note "3" on page 189, below).

Note:

1. EventMonitorCreate returns a monitor token that EventWait uses to identify the asynchronous requests it is waiting for. There are two ways an application can call EventMonitorCreate:
   - Call EventMonitorCreate for each event key. EventMonitorCreate returns a unique monitor token for each key. A call to EventWait with that token causes the thread to wait for the completion of the request for that file pool and work unit.
   - Call EventMonitorCreate for an array of unique event keys. EventMonitorCreate returns a single monitor token. A call to EventWait with that token causes the thread to wait for the completion of a request matching one of the event keys. The application can determine which asynchronous request has been completed by testing the event flag values returned by EventWait.
2. When an asynchronous CSL routine returns a 1 for the request ID, the request was completed synchronously. Do not call EventWait.
3. When DMSCHECK is issued with the NOWAIT option, the application may need to issue EventWait and DMSCHECK more than once before it receives confirmation that the request has been completed.

See *z/VM: CMS Application Multitasking* for information on Event Management.

Sharing SFS Files Across Systems

Users may specify a nickname on SFS commands which accept a user ID, such as GRANT AUTHORITY. However, the user may have to set up the nickname differently for files shared across systems. One reason for this is that the node ID is not recognized. Only user IDs are passed. This necessitates the rule that all user IDs within a TSAF or CS collection be unique.

With cross-collection support, which involves an APPC/VM VTAM® Support (AVS) virtual machine, maintaining unique user IDs across the network is not practical. Therefore, each user who needs resources in another collection is assigned a local user ID in that collection. The AVS virtual machine then maps the user ID and where it came from to the local ID, which is administered to be unique within this collection.

If the nickname maps to a user ID and node ID, CMS does not know where the node is. If it knew that the node was within the collection, then the node ID could be ignored. However, it does not know that. Also, from outside the collection, the local user ID is used, not the user ID/node ID.

Therefore, SFS looks for a user-specified tag in the nickname called LOCALID. If found, it is used to resolve the nickname instead of the user ID. The LOCALID itself may not be a nickname. The LOCALID value may be a list of user IDs. If a LOCALID tag is specified, it will replace the USERID, NODE, and List Of Userid tags. Obviously, LOCALIDs may not contain node IDs.

A user who is setting up a nickname for another user will do the following: If the other user is on the same system, then there is no change for the current nickname entry. Either no node ID is specified or this node is specified. CMS recognizes the node ID for this machine. If the other user is on another machine within the collection, then a user ID and node ID are specified for use by RSCS and other programs which use node ID. A LOCALID tag with a value of the other user ID without the node ID is added. Because all user IDs within a collection are unique, the real user ID is ok. If the other user is on a machine outside the collection, the user ID/node ID are specified for programs which use node ID. For SFS, the LOCALID, which was assigned by an administrator, and specified to the AVS virtual machine, is added.

The steps for SFS command nickname resolution are:

1. If a LOCALID tag is specified, use it.
2. If no node IDs are specified for the Userid tag or the List Of Userids tag, then use those user IDs.
3. If node IDs are specified, and represent this machine, then use the user IDs specified.
4. If any node IDs are specified which represent other nodes, check if a nickname exists for that user ID/node ID. If the nickname exists, and a LOCALID exists for that nickname, use the LOCALID.
5. If none of these cases exist, an error is returned.

You can use the DMSQCONN routine (Query Connect) to determine if an SFS server is at a local or remote location.

Use of APPC/VM Paths by SFS

SFS uses APPC/VM to communicate between your virtual machine and the virtual machine that manages the SFS file pool.

A path is established when you use a unique work unit ID and SFS file pool ID combination. An existing path to the correct SFS file pool will be used if there is no unit of work in process on that path. For example, in Figure 25 on page 190, Ernie runs a program that uses two work units. In Work Unit A, Ernie refers to files in both the DEVELOP and PROD SFS file pools. So, for Work Unit A, two paths are established: one to SERVER1, and another to SERVER2.

Ernie's program also uses a second work unit that refers to files in the DEVELOP SFS file pool. Because the references to the files are in another work unit, Work Unit B, another path to SERVER1 is used. Otherwise, Ernie would not be able to commit the work independently.

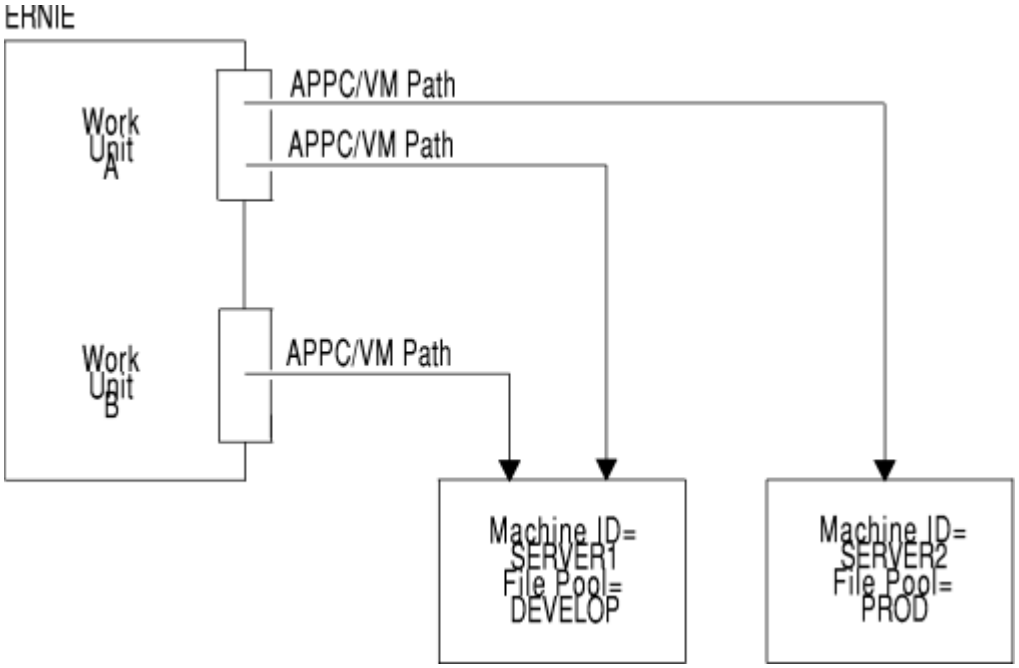


Figure 25. Use of APPC/VM Paths by SFS

Use of APPC/VM Paths with the Default Work Unit ID

In the previous figure, either Work Unit A or B could be the default work unit ID. Paths established for the default work unit ID normally persist until you log off. If the default work unit is again used during the same CMS session, CMS will try to reuse an existing path. Usually it can, because most CMS users issue commands against only a few SFS file pools.

For example, CMS commands are executed using a default work unit ID and usually a default SFS file pool. When the command completes, the work is either committed or rolled back, but the path persists. If the user executes another command, CMS uses the same default work unit ID, SFS file pool, and path. If the user next refers to a file in another SFS file pool, a new path is established for that work unit ID. CMS will reuse the new path for any future requests to the same SFS file pool.

Use of APPC/VM Paths with Acquired Work Unit IDs

A work unit ID you get with DMSGETWU persists until you re-IPL or log off, at which time all paths associated with that work unit ID are severed. A nondefault work unit ID can be made the default work unit ID temporarily. You can change your default work unit ID by using the DMSPUSWU routine. You can reset it to the previous value by using DMSPOPWU.

Commands executed in CMS Subset mode do not use the same work unit ID as commands executing when not in Subset mode. Instead, CMS provides a different default work unit ID.

Severed APPC/VM Paths in SFS

All SFS paths from your virtual machine are severed when you log off, re-IPL CMS, or issue the DMSPURWU routine (Purge Work Unit IDs). Whether you log off, re-IPL, or issue the DMSPURWU routine to sever a path, there is additional overhead to reconnect if you later want to reestablish paths to the SFS server. There are other ways that your paths can be severed which are not under your direct control:

- The SFS file pool server machine operator forces you off the file pool (issues the FORCE USER server operator command).
- An APPC/VM communication error occurs.
- CRR processing has to sever one or more paths to ensure a complete backout, to enable resynchronization to occur, or both.
- File pool server processing ends, either normally or abnormally. (In this case, all paths that the server maintained are severed.)
- CMS end-of-command processing takes place and there are pending asynchronous requests.

If a path is severed when communication is outstanding, the outstanding file pool request will fail with an error code indicating that the path was severed. If a work unit is in progress on that path, the work unit is rolled back.

If a path is severed asynchronously (when there is no communication outstanding) and a work unit was in progress on that path, your next request to that file pool for that work unit will fail with an error code indicating the path was severed, and the work unit is rolled back.

If a path is severed asynchronously and a work unit was not in progress on that path, your next request to that file pool for that work unit will succeed, assuming the server is now available.

While paths are established from machine-to-machine, you should remember that users request files from SFS file pools, not SFS file pool server machines. That is, the user specifies an SFS file pool ID, not the ID of the virtual machine that happens to be managing the file pool. The SFS file pool ID (resource ID) is used in the APPC/VM communications. APPC/VM finds the machine that is managing the file pool. This machine can be on the same system as the requester, another system in the TSAF or CS collection, or another system in an SNA network.

Because users and applications request files from an SFS file pool, not a SFS file pool server machine, one wonders what might happen to the paths if the SFS file pool was switched to another SFS file pool server

machine. That is, suppose the SFS file pool is switched from one SFS file pool server machine during lunch, when many user machines are still in a CMS session but, perhaps disconnected.

To switch SFS file pool server machines, the original machine would have to be shut down. All its links would be severed, which does not interfere with any work (everyone's eating lunch). The minidisks and appropriate files are assigned to the new SFS file pool server machine. It is started using the same file pool ID as the first machine.

Now suppose a user returns from lunch, reconnects, and invokes XEDIT for a file in the SFS file pool. What happens? If a work unit was in-progress when the original machine was shut down, CMS sees that the path is severed and returns an error indication. If you immediately reaccess the file pool and then reissue the XEDIT command, CMS establishes a new path using the SFS file pool ID. APPC/VM communications find the new virtual machine, and completes the path to it. If a work unit was not in-progress when the original machine was shut down, the old link is severed, a new one is established, and the command completes successfully.

Chapter 13. Manipulating BFS Files and Directories Using CMS Record File System CSL Routines

OpenExtensions Byte File System (BFS) files are stored in CMS file pools. A byte file system is generated as a file space in a file pool, and several byte file systems can reside in the same file pool. The CMS record file system interface supports limited manipulation of BFS files and directories, primarily for administration and system-managed storage.

Special Meanings for File and Directory in This Context

In the context of manipulating BFS objects in the CMS record file system, the term "file" refers to a BFS regular file only. Other types of BFS files cannot be manipulated by CMS record file system functions. The term "directory" refers to only 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.

For information on how CMS file attributes apply to BFS files, see Chapter 11, “Understanding the CMS File System,” on page 119. Many of the CMS functions described in Chapter 12, “Manipulating SFS and Minidisk Files and Directories,” on page 129 apply to BFS files and directories.

This chapter primarily describes where the CMS record file system support for BFS is different. Topics include:

- Programming interfaces
- Authority required
- Storage management by DFSMS/VM
- Application design considerations
- File I/O
- Directory I/O
- File locking
- Use of file pool space.

Programming Interfaces

To manipulate BFS files and directories, CMS provides a call interface composed of a set of CSL routines located in the VMMTLIB callable services library. (This library resides in the CMS nucleus.) These routines, known as the OpenExtensions callable services, are intended for use by language run-time environments. They can also be called from assembler and REXX programs. For more information about these routines, see the *z/VM: OpenExtensions Callable Services Reference*.

CMS Pipelines provides stage commands for manipulating BFS files and directories. For more information, see the *z/VM: CMS Pipelines User's Guide and Reference*.

The CMS record file system CSL routines in the VMLIB callable services library provide limited function for manipulating BFS files and directories. (The FS macros do not provide BFS support.) When used for BFS objects, the CMS record file system interface has the following general characteristics:

- It accepts file names, directory names, and name definitions (namedefs).
- It automatically commits changes at file close time.
- It does not participate in Coordinated Resource Recovery (CRR).

For general information about calling the CSL routines in VMLIB, see “CMS Record File System Programming Interface” on page 129.

Required Authority

Using the CMS record file system interface to manipulate BFS files and directories requires file pool administrator authority. Note that CMS functions such as the QUERY ENROLL command, which do not actually *manipulate* BFS objects but only provide information about them, do not require administrator authority.

DFSMS/VM and BFS File Management

DFSMS/VM can help automate storage management tasks for BFS files as it does for SFS files. Files can be assigned attributes that tell DFSMS/VM how long to maintain the file. Files can be automatically deleted or moved to DFSMS/VM-owned storage automatically, to more efficiently use the available storage.

You may want to ask your file pool administrator if DFSMS/VM has been installed on your system and is being used to manage SFS and BFS files, because this can affect the behavior of files in a file pool. See *z/VM: DFSMS/VM Planning Guide* for an explanation of how this product is used.

Migration and Recall

Some BFS files that appear to reside in your file pool may actually have had their data moved into a storage repository managed by DFSMS/VM. (Note that such files still are considered by BFS to consume their usual amount of room in your file space.) These files are said to be in ***migrated status*** in your file pool. You can identify files that have been placed in migrated status by using the OPENVM LISTFILE command with the ATTRIBUTES option (the default). Files in migrated status have an F* in the Type column of the response.

Migrated files behave exactly like regular BFS files, but they must be recalled (either automatically or explicitly) into your actual file pool before you can reference the data. This may cause a delay, depending on your system configuration and workload. Automatic recall is governed by the CMS SET RECALL command. If SET RECALL is ON (the default), recall happens automatically when the file data is referenced. If SET RECALL is OFF, the file is not recalled. You receive an error indicating that the file is migrated and not available. You can add the SET RECALL setting to your PROFILE EXEC. Explicit recall is performed with the DFSMS RECALL command. (See *z/VM: CMS Commands and Utilities Reference* for more information about SET RECALL and *z/VM: DFSMS/VM Storage Administration* for information about DFSMS RECALL.)

A file does not need to be recalled unless you need to access the file data itself (for example, with the XEDIT command or the DMSOPEN routine with an intent of READ).

Automatic File Movement and Erasure

You should be aware that DFSMS/VM can automatically cause BFS files to be placed in migrated status (that is, move the file data into its storage repository) or erased at certain predetermined times, without advance warning, according to your installation's storage management policies. File erasure criteria are usually related to how long the file has existed, or the length of time since it was last referenced. The entire file may be erased, or only the data in the file. See *z/VM: DFSMS/VM Storage Administration*.

Application Design Considerations

When designing your application program to manipulate BFS files and directories, consider the following:

- Use a name definition (namedef) to identify a file or directory to your program.
- Acquire work unit IDs for work units.
- Process file pool requests on behalf of other user IDs.
- Be aware that BFS changes are committed when closed.
- Handle unexpected conditions in your program.

Using a Namedef

For any CSL routine that requires a file name and file type, directory identifier, or some combination of these in its parameter list, you can specify a namedef instead. A namedef is a 1- to 16-character string that represents either:

- A file name and file type. For a BFS file, this is the system-generated numeric CMS file name and file type.
- A directory ID. For BFS, this is the name of the byte file system (*bfsid*), which identifies the top directory in a BFS file space.

By using namedefs in your parameter lists, you can run a program to process different files and directories without changing the code and recompiling the program. The file and directory names are defined externally to the program.

To associate a namedef with the name of a file or directory, you issue a CREATE NAMEDEF command before running the program. The first character of the namedef must be alphabetic, and the remaining characters must be alphabetic or numeric. When the program runs, CMS does the operations on the file or directory that the namedef represents.

You can use namedefs in three different ways depending on the operations that your program is going to perform.

- If your program uses different files within the same directory, you could use a namedef for the file name and file type and code the directory name directly in the parameter list. This lets you process different files in the same byte file system.
- If your program uses the same file name and file type but different directories, code the file name and file type directly but use a namedef for the directory name. This lets you process separate versions of the same file that reside in different byte file systems.
- For the most flexibility, use namedefs for the file name and file type and for directory name. In this case, the program can process any file name and file type in any byte file system.

Because a namedef is resolved at run time, you do not need to have them defined before compiling your program. If you run a program without defining the namedef, however, the program may fail because the namedef is not defined.

A namedef continues until the end of the CMS session (or until an abend occurs) unless you change the definition of the namedef or delete it altogether.

Creating a Namedef

Suppose you code your parameter list using the namedef FNAME for the file name and file type and DIRNAME for the directory ID. To process the file having a system-generated file ID of 2 0 in byte file system POOLA:BFS1, the user would have to enter two CREATE NAMEDEF commands before running your program:

```
create namedef 2 0 fname
create namedef poola:bfs1 dirname
```

To change the definition, you would enter a CREATE NAMEDEF command with a REPLACE option. For example, to change FNAME to refer to file 17 0, you would enter:

```
create namedef 17 0 fname (replace
```

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

Deleting a Namedef

To delete the namedef, enter a DELETE NAMEDEF command:

```
delete namedef fname
```

To delete all namedefs you have defined in this CMS session, enter:

```
delete namedef *
```

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

Additional Considerations for Directory ID

Many file pool commands and routines that operate on BFS objects require directory identifiers as a part of their syntax and parameter lists. CMS allows the user or application to let the file pool ID and file space ID portions of the directory identifier default. For example, in the command `CREATE LOCK .`, the file pool ID and file space ID have not been specified. CMS will attempt to fill in the file pool ID and file space ID values.

If the file pool ID is not specified, the default file pool set with the SET FILEPOOL command is used. A default file pool for a user can also be specified in the CP directory entry for that user. If neither of these have been set, the command or routine fails, because the system does not set a default file pool ID. If the file space ID is not specified, it defaults first to the file space ID set with the SET FILESPACE command, and then to the user ID calling the routine.

Using Work Units in Application Programs

A ***work unit*** identifies related file pool requests. A work unit is identified by a fullword number called a ***work unit ID***. An application can have many active work units at any given time. Unlike SFS files, where data can be committed whenever appropriate for your application, changes to BFS files are committed when the files are closed.

A work unit can be associated with multiple file pool servers or multiple work units can be associated with one file pool server. In addition, other resource managers can be accessed on the same work unit as SFS. However, updates to SFS files and BFS files in the same file pool cannot be included on a single work unit.

A work unit can also be associated with a specific file space or user ID by using the DMSGETWU CSL routine. A service virtual machine with administration authority can issue file pool requests on behalf of disconnected user IDs, or user IDs that differ from the VM ID of the service virtual machine, and all SFS authorizations will remain in force.

Using Multiple Work Units in a Program

To achieve greater system performance and throughput, a program should open BFS files or directories at the last possible moment, and should close the files as soon as possible. By doing so, the program uses the least amount of 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, as these changes cannot be combined on a single work unit.

For more information about using work units, see “Using Work Units in Application Programs” on page 133.

Committing and Rolling Back Changes in Application Programs

Changes to a BFS file are committed when the file is closed. The changes cannot be committed before the file is closed, and the file cannot be closed without committing the changes. The COMMIT parameter is ignored if specified on any CSL routine except DMSCLOSE. Likewise, the NOCOMMIT parameter on the DMSERASE and DMSCLOSE routines has no effect. However, DMSROLLB can be used to roll back changes before the file is closed. The recoverability and overwrite attributes of a BFS file are always RECOVER and NOTINPLACE. These attributes cannot be changed.

Handling Unexpected Conditions

When a program encounters an unexpected condition, it can do one or more of the following:

- Terminate processing

- Set a return code
- Issue a message and return code to the user.

Collecting Error Information

The CSL routines that support BFS file and directory manipulation provide three sources of error information:

- Return codes
- Reason codes
- Workunit extended error information.

The first two sources are required parameters for every routine; the last source is optional.

The ***return code*** provides general error information for a routine. The return code is placed in the return code variable that you provide and in general register 15. (The return code variable is a signed fullword.) These CSL routines may have one of the following return codes:

0
Normal—the routine completed successfully.

4
Warning—the routine completed, but the result may not be as intended.

8
Error—the routine did not complete.

12
Error—the routine did not complete, and work within the work unit ID was rolled back.

You may also receive return codes from DMSCSL that are negative values. For more information on these return codes, see the *z/VM: CMS Callable Services Reference*.

For many conditions where you receive a return code of 8, the state of the work unit stays the same.

Return codes 4, 8, and 12 have ***reason codes*** associated with them to further describe the warning or error condition. The reason code for a routine is placed in the reason code variable that you provide and in general register 0. (The reason code variable is a signed fullword.) Return code 0 does have an associated reason code of 0. For a list of the return codes and associated reason codes, see the *z/VM: CMS Callable Services Reference*.

Workunit extended error information (*wuerror* parameter) is an optional source of error information. It contains additional error information from the file pool. On input, the *wuerror* parameter is a character string followed by a length parameter. If you omit it or if the variable has a length field with a value of 0, only the return code and reason code are returned. The following information is returned by *wuerror*:

- Length of the *wuerror* parameter
- Number of file pool error information areas returned
- Total number of errors for which information is available
- One or more groups of file pool error information.

The file pool error information contains information about errors that have occurred. This information includes: error reason codes, warning reason codes (up to 16 possible), and user ID index. The DMSWUERR routine converts the *wuerror* output to data placed in individual variables. The WUERROR and FPERROR macros let routines map into the work unit (WUERROR) and file pool (FPERROR) data areas. For more information, see the *z/VM: CMS Callable Services Reference* and the *z/VM: CMS Macros and Functions Reference*.

BFS File I/O

This section describes how to use CMS record file system CSL routines to manage BFS files. The routines that let you manage files are collectively known as File I/O routines. The File I/O routines that you can use for BFS files are listed in Table 21 on page 198.

Note: You cannot create new BFS files using this interface. You can manipulate only existing BFS files. BFS files can be created only through the OpenExtensions interface.

Table 21. CMS Record File System CSL Routines for BFS File I/O

CSL Call	CSL Function	Description
DMSCLOSE	Close	Closes a file (logically disconnects an application program from a specific file).
DMSCLDBK	Close Blocks	Closes a file for reading and writing data blocks (logically disconnects an application program from a specific file).
DMSERASE	Erase	Erases files.
DMSEXIFI	Exist - File	Checks if a file exists and returns the file information in variables.
DMSEXIST	Exist	Checks if a file (or directory) exists and returns the file information in a buffer.
DMSOPDBK	Open Data Blocks	Opens a file for reading and writing file data blocks (logically connects an application program to a specific file).
DMSOPEN	Open	Opens a file (logically connects an application program to a specific file).
DMSPOINT	Point	Alters the read and write record pointers in a file opened by DMSOPEN.
DMSRDDBK	Read Data Blocks	Reads one or more file data blocks.
DMSREAD	Read	Reads one or more records from a file.
DMSROLLB	Rollback	Rolls back changes to an open file.
DMSVALDT	Validate	Checks the validity of a file identifier.
DMSWRDBK	Write Data Blocks	Writes one or more file data blocks.
DMSWRITE	Write	Writes one or more records to a file.

Determining If a BFS File Exists

To determine if a BFS file exists and get the status of the file, you can use the DMSEXIFI or DMSEXIST routine. However, if you plan to read or write to the file ***and*** share the file with other users, you should not bother with determining if the file exists. You should begin file I/O by opening the file with an intent other than READ. This ensures that no other users could erase or revoke the file before you get a chance to open the file. Once you have the file opened, other users cannot erase the file.

DMSEXIFI returns file information in variables. If the file does not exist or you are not authorized to read from the directory, DMSEXIFI returns a return code of 8 and a reason code of 30000 or 44000. If the file does exist, DMSEXIFI returns a return code of 0 and places the file information in the variables that you provide. Check the authority information passed back to make sure you have the correct authority.

DMSEXIST returns file information in a specified buffer. If the file does not exist or you are not authorized to read from the directory, DMSEXIST returns a return code of 8 and a reason code of 90220. If the file does exist, DMSEXIST returns a return code of 0 and places the file information in the specified buffer. Check the authority information passed back to make sure you have the correct authority.

You can map the output with the EXSBUFF assembler mapping macro. For more information on EXSBUFF, see the *z/VM: CMS Macros and Functions Reference*.

Opening BFS Files

To open a BFS file (that is, establish a logical connection to the file for subsequent reading or writing of records, or both), use the DMSOPEN routine. When you use DMSOPEN, you specify an intent to

indicate the type of operation that you are performing. You can also specify the type of I/O that you want performed.

You indicate the intent by specifying one of the following parameters:

- **READ** means that the file will only be read. You cannot open a file with READ if it does not exist.
- **REPLACE** indicates that if the file exists, you will replace it with only the added records. You cannot open a file with REPLACE if it does not exist. When you have opened a file for replace, you can read only records that you have written. Attempting to read records before writing any results in an end-of-file condition (return code = 4).

  When a file is replaced, the old version of the file is shadowed by the SFS server. If no records are written to the file or you do not want to actually replace it, this operation can be rolled back later on.

You can indicate the type of I/O by specifying one of the following parameters:

- **CACHE** should be specified when the caller intends to read the data sequentially most of the time. Specifying this parameter causes the file system to cache several data blocks for the file, performing I/O only when the cache buffer is full (for writing) or empty (for reading). This generally reduces the number of separate I/O operations performed on the file.
- **NOCACHE** should be specified when the caller intends to read the data in random order most of the time.

If you do not specify CACHE or NOCACHE, the system chooses a method that it considers appropriate.

A BFS file always has attributes of RECOVER (meaning uncommitted changes are backed out as the result of a system initiated rollback) and NOTINPLACE (meaning the reader sees a consistent version of the file from open to close).

When you open a file, CMS passes a ***token*** back to your program. The token is an 8-byte field that identifies the file. You will pass this token on to other routines for reading, writing, and closing the same file.

After a file is open, you may need to determine certain attributes of the file, such as the number of records in the file, or the date and time the file was last modified. You can use the extract function of the DMSERP routine to obtain this type of information. See the *z/VM: CMS Callable Services Reference* for more information on DMSERP and the extract functions that are available.

Reading and Writing BFS Files

Use the DMSREAD and DMSWRITE routines to read and write BFS files in the same manner as CMS record files. For more information, see “Reading and Writing Files” on page 154.

Closing BFS Files

Use the DMSCLOSE routine to close BFS files previously opened with DMSOPEN. When a BFS file is closed, all changes are committed, even if you specify NOCOMMIT.

Erasing BFS Files

Use the DMSERASE routine to delete a BFS file. If the ENTIRE parameter is used (the default), the file and all links are erased. If the DATAONLY parameter is specified, only the contents of the file are deleted; the links remain.

Note: In BFS, a ***link*** is a new path name to an existing file, similar in concept to an SFS alias. For more information about BFS links, see the *z/VM: OpenExtensions User's Guide*.

You cannot erase a BFS file if:

- The file is not a BFS regular file.
- The file is open.
- The file is locked by another user.

- You are not a file pool administrator.

Data Block I/O

The data block interface can be used to transfer data to and from BFS files. For more information, see "Data Block I/O" on page 163. Note that data being transferred to a BFS file *must* have a logical record length of 1, and that data transferred from a BFS file has a logical record length of 1.

BFS Directory I/O

This section describes how to use the CMS record file system CSL routines to manage BFS top directories. The CSL routines you can use for BFS directory I/O are listed in Table 22 on page 200.

Note: You cannot create new BFS directories using this interface. You can manipulate only existing BFS directories. BFS directories can be created only by the OpenExtensions interface.

Table 22. CMS Record File System Routines for BFS Directory I/O

CSL Call	CSL Function	Description
DMSCLDIR	Close Directory	Logically disconnects an application program from a specific directory.
DMSEXIDI	Exist - Directory	Checks for an existing directory and returns the directory information in variables.
DMSEXIST	Exist	Checks for an existing directory (or file) and returns the directory information in a buffer.
DMSGETDD	Get Directory - Dir	Reads one or more directory records into variables when a directory has been opened with an intent of DIR.
DMSGETDI	Get Directory	Reads one or more directory records into a buffer.
DMSGETDK	Get Directory - Lock	Reads one or more directory records into variables when a directory has been opened with an intent of LOCK.
DMSGETDL	Get Directory - Alias	Reads one or more directory records into variables when a directory has been opened with an intent of ALIAS.
DMSOPDIR	Open Directory	Logically connects an application program to a specific directory for subsequent reading.

Here are some of the tasks you can perform on BFS directories:

- Determine if the directory exists — You can use the DMSEXIST or DMSEXIDI routines to determine whether the directory you want to work with exists. You may want to skip this step and just open the directory, because the status of the directory could change from the time you determine its existence to the time you actually open it.
- Open a directory to read it — Use the DMSOPDIR routine to logically connect your program to a specific directory. You can open the directory more than once.
- Read directory records — The Get Directory routines allow you to scan files by reading records in the directory (previously opened using DMSOPDIR). You can also scan for lock information. These routines allow you to get the system-generated CMS file names for BFS files.
- Close a directory — The DMSCLDIR routine logically disconnects your application program from a specific directory (previously opened using DMSOPDIR).

Opening BFS Directories

You can determine the existence of a BFS directory by using the DMSEXIDI or DMSEXIST routine. However, as with SFS directories (see "Determining If an SFS Directory Exists" on page 167), it is probably

better to begin directory I/O by opening the directory using the DMSOPDIR routine. You specify an intent on DMSOPDIR to indicate the type of operation that you will perform. This determines the type of information that will be available on subsequent Get Directory routines. Intents of FILEEXT, LOCK, DIR, and ALIAS are permitted.

Note: The DIR and ALIAS intents do not perform any useful function for BFS. They are supported only for compatibility with existing programs that use these intents.

FILEEXT: With this intent, you must specify a file name and file type (system-generated values or asterisks) or a namedef. Information returned from later DMSGETDI and DMSGETDX calls includes:

- Type of open = 8 (FILEEXT)
- File space type = 1 (BFS)
- File ID = system-generated file name and file type
- File mode number = 1
- Record format = F (fixed)
- Recoverability attribute = 1 (RECOVER)
- Overwrite attribute = 0 (NOTINPLACE)
- Logical record length = 1 (1 byte)
- Number of blocks
- Number of records = number of bytes in the file or -1 (for greater than 2^{31}-1 bytes)
- Date of last update
- Time of last update
- Directory attribute = 0 (BFS top directory)
- DFSMS/VM migrated status indicator
- User ID of the owner = name of the BFS file space
- Creation time
- Status = 1 (BFS regular file)
- Authority
- Date of last reference
- Creation date
- Maximum blocks used for the file
- Data blocks used for the file
- System blocks = number of additional blocks generated by the system
- DFSMS/VM related attributes
- Unique ID
- Last change date
- Last change time.

For the exact output format for an intent of FILEEXT, see the DMSGETDI or DMSGETDX routine in the *z/VM: CMS Callable Services Reference*.

LOCK: With this intent, specifying a file name and file type (system-generated values or asterisks) or a namedef is optional. Information returned from later DMSGETDI and DMSGETDK calls includes:

- Type of open = 6 (LOCK)
- File space type = 1 (BFS)
- File ID = system-generated file name and file type
- File mode number = 1
- Status = 1 (BFS regular file)

- Lock type = 2 (exclusive)
- Lock length = 1 (session) or 2 (lasting)
- Lock user ID
- Directory attribute = 0 (BFS top directory)
- DFSMS/VM migrated status indicator.

For the exact output format for an intent of LOCK, see the DMSGETDI or DMSGETDK routine in the *z/VM: CMS Callable Services Reference*.

DIR: This intent is supported for compatibility with existing programs that use this intent. Only a BFS top directory name or a namedef can be specified. Information about BFS subdirectories is not returned. Information returned from later DMSGETDI and DMSGETDD calls consists of:

- Type of open = 7 (DIR)
- Directory length
- Directory name
- Directory attribute = 0 (BFS top directory).

For the exact output format for an intent of DIR, see the DMSGETDI routine in the *z/VM: CMS Callable Services Reference*.

ALIAS: This intent is supported for compatibility with existing programs that use this intent. BFS files do not have aliases. A file name and file type (system-generated values or asterisks) or a namedef can be specified. The information returned on later DMSGETDI calls indicates that the specified file has no aliases.

For the exact output format for an intent of ALIAS, see the DMSGETDI routine in the *z/VM: CMS Callable Services Reference*.

Reading BFS Directories

Once the directory is open, use the Get Directory routines to read one or more records:

- Use DMSGETDI to return the information in a buffer.
- Use DMSGETDD to return the information in variables if you opened the directory with an intent of DIR.
- Use DMSGETDX to return the information in variables if you opened the directory with an intent of FILEXT.
- Use DMSGETDK to return the information in variables if you opened the directory with an intent of LOCK.

For more information, see “Reading Directories” on page 170.

Closing BFS Directories

Use the DMSCLDIR routine to close a directory that was previously opened using DMSOPDIR. Closing a directory logically disconnects your program from the directory so that you can no longer read records in it. You identify the directory by passing the token returned from DMSOPDIR.

Erasing BFS Directories

You cannot use the DMSERASE routine to erase a BFS directory. If you really want to erase the directory (remove the byte file system from the file pool), use the DMSDEUSR (Delete File Space) routine.

Locking BFS Files

This section describes how file pool locking applies to the BFS files stored in file pools. The following routines provide file locking:

CSL Call	Function	Description
DMSCRLOC	Create Lock	Creates an explicit lock on a file.
DMSDELOC	Delete Lock	Deletes an explicit lock on a file.

Note: You cannot use DMSCRLOC to lock a BFS top directory. To lock an entire byte file system, use the DMSDISFS routine to disable the BFS file space.

When more than one person can write to a file, there must be some way to ensure that two people do not update the same file at the same time. The file pool locking mechanism provides as much concurrency as possible. To prevent simultaneous updates, the file pool server uses a locking scheme composed of implicit and explicit locks.

An ***implicit lock*** is one that the server acquires and releases automatically. Whenever someone has a file open, the server internally associates an implicit lock with that object. If someone else tries to read or write to that file, the server first checks whether the file is locked before allowing access. An ***explicit lock*** is one that you create by issuing a CMS command or routine that forces a file to be locked.

Locks have two important characteristics: duration and type. The ***lock duration*** is the length of time the lock is to exist. The ***lock type*** determines whether others can share the file during the time it is locked.

Implicit Locking

Implicit locks are automatically acquired by the server to ensure data integrity among multiple users. To allow greatest concurrency, the server acquires locks only when needed and frees them as soon as it can. The lock duration is the length of a server unit of work, which is, for example, from the time a file is opened until it is closed and any changes are committed. Maintaining the implicit lock eliminates the need to obtain an explicit lock each time a file is updated.

Internally, the server uses several different lock types. For the purposes of this discussion, however, it is best to think of implicit locks as having only two types:

SHARE
: An implicit share lock permits multiple readers of a file. The server acquires an implicit share lock when a user opens a file for read. When a file is implicitly share locked, other users can implicitly lock the file as share or exclusive.

EXCLUSIVE
: An implicit exclusive lock permits only one writer of a file. The server acquires an implicit exclusive lock when a user opens a file for anything other than read. When a file is implicitly exclusive locked, other users can implicitly lock the file as share, but they cannot implicitly lock the file as exclusive.

The lock type and duration for implicit locks depends on the operation you ask the server to perform. For instance, the server gets different locks for a replace request (DMSOPEN routine with the REPLACE parameter) than it does for an erase request (DMSERASE). Some locks can be freed when the operation completes, while other locks must be held until the work unit in which the operation occurs is committed. In any case, all implicit locks acquired during a work unit are freed when the work unit is rolled back. Explicit locks, which we will be discussing next, can last beyond the work unit (even if there is a rollback).

When you try to update a file that is locked with an implicit exclusive lock, your request is rejected. If you made the request in a program, your program receives a return code of 8. If you made the request by issuing a CMS command, the command fails.

Explicit Locking

When you let the file pool server automatically lock and unlock your BFS files, the implicit lock on a file remains in effect until the file is closed and the changes have been committed. If you want a lock to remain in effect longer than this, you must explicitly lock the file by entering a CREATE LOCK command before running your program or by calling the DMSCRLOC (Create Lock) routine within your program. This section discusses using the DMSCRLOC routine. For more information on the CREATE LOCK command, see the *z/VM: CMS User's Guide* and the *z/VM: CMS Commands and Utilities Reference*.

The DMSCRLOC routine allows you to explicitly lock a BFS file. Explicit locks are sometimes called *check-out* locks because you are checking out a file just as you would check out a book at a library. With the DMSCRLOC routine you can specify both the duration of the lock and the type of lock.

You can specify the following types of explicit locks for BFS files:

EXCLUSIVE
An explicit exclusive lock type means there can be only one person accessing the file at a given time. If you create an exclusive lock, you are the only person that can read or write the file. It also prevents other users from getting any locks on the file. To obtain an exclusive lock, there cannot be any other types of locks on the file.

UPDATE
An explicit update lock type has the same effect as an explicit exclusive lock.

The duration of the lock can be:

SESSION
An explicit session lock lasts until the end of the CMS session, or until it is specifically deleted (with the DELETE LOCK command or DMSDELOC routine), or until all user machine connections to the file pool are broken (for example, with the DMSPURWU routine).

LASTING
An explicit lasting lock lasts until it is deleted with the DELETE LOCK command or DMSDELOC routine. The lock lasts across CMS sessions and logon sessions. For example, if you create a LASTING lock on a file and then log off, the file is still locked even though you are not logged on.

You should use explicit locks whenever you want to control the activity on your files without revoking permissions. If, for example, you are rewriting a document that everyone has access to, and you do not want anyone to see the new draft until it is complete, you might create lasting exclusive locks on the document's files.

To request a lock, all activity in the affected file pool for the work unit must be committed. If there is any outstanding work on the file pool, the request fails.

Relationships between Locks

Table 23 on page 204 depicts the relationships between the various locks on BFS objects. Remember the following rules when considering locks:

- Implicit locks are acquired by the file pool server.
- Explicit locks are acquired by users (or by some CMS commands like XEDIT or SET LANGUAGE).
- An explicit exclusive lock has no readers and one writer.
- To obtain any type of lock, there cannot be any other types of locks on the file or directory.

Table 23. Results of Interactions between Accessing and Locking BFS Objects

		when you try to		
		Read	**Write**	**Create an Exclusive Lock**
If someone is	reading	OK	OK	fail/wait
	writing	OK	fail/wait	fail/wait
If someone has already created	an exclusive lock	fail	fail	fail

Table 23. Results of Interactions between Accessing and Locking BFS Objects (continued)

	when you try to		
	Read	**Write**	**Create an Exclusive Lock**

Legend:

OK
Done immediately.

fail
Operation fails or program receives an error code.

fail/wait
Indicates action with FILEWAIT ON or OFF, which applies only to implicit locks. See *z/VM: CMS Commands and Utilities Reference* for information on the SET FILEWAIT command.

Table 24 on page 205 shows the interactions between implicit locks when the same BFS object is accessed from both the OpenExtensions interface and the CMS record file system interface.

For example, if a user is using the OpenExtensions interface to read or write a BFS object, CMS record file system users cannot get write access to the object. If a user is using the CMS record file system interface to write a BFS object, OpenExtensions users are denied both read and write access to the object.

Note: The interactions shown in Table 23 on page 204 and Table 24 on page 205 do not apply to CMS objects defined in BFS by external links. In that case, the interactions shown in Table 16 on page 179 apply.

Table 24. Lock Interactions between the OpenExtensions and CMS Interfaces

		when you try to access the object from			
		OE		**CMS**	
and someone already has the object accessed from		**Read**	**Write**	**Read**	**Write**
OE	**Read**	OK	OK	OK	fail
	Write	OK	OK	OK	fail
CMS	**Read**	OK	OK	OK	OK
	Write	fail	fail	OK	fail

Legend:

OE
OpenExtensions interface.

CMS
CMS record file system interface.

OK
Operation is done.

fail
Operation fails or program receives an error code.

Deleting Locks

The DMSDELOC (Delete Lock) routine releases an explicit lock on a file that was created with the CREATE LOCK command or DMSCRLOC routine. Only the creator of the lock or a file pool administrator can delete it.

To delete a lock, all activity in the affected file pool for the work unit must be committed. If there is any outstanding activity, the request fails. If you want to delete a lock in an active file pool, you can use another work unit. For example, say there is an open file in file pool A, which is active in the default work unit, and you want to delete the lock on that file. You can call DMSGETWU to get another work unit ID and then call DMSDELOC (on the new work unit ID) to delete the lock in file pool A.

Waiting for Locks

If your request fails because the file is locked, you should wait until the file is not locked and reissue the request. A program that fails because of lock conflicts should be rerun when the files it needs are not locked.

If you do not care how long you might wait for a response to a request, and if you are being rejected because you are the second writer, you can tell the server to wait for the file to become unlocked rather than reject the request. To do this, you issue a SET FILEWAIT ON command. However, this can help only if the files are locked implicitly.

This is useful, for example, at the end of the day, when you might start a program, disconnect your virtual machine, and then go home. It is not a good idea to submit a job to the batch machine that issues a SET FILEWAIT ON command. While your program is waiting, so are all the other programs in the batch machine queue.

Because most CMS commands issued from a terminal form a single logical unit of work, the implicit lock is usually not held for a long time. In application programs, however, you have control over the committing of work units and can cause implicit locks to be held for a long time.

Deadlocks

In any system that manages shared resources, it is possible for deadlocks to occur. A ***deadlock*** is a standstill that is reached when two or more users are each waiting for a resource that the other holds.

In a CMS file pool, it is possible for deadlocks to occur only for implicit locks. Because the file pool server never waits for an explicitly locked object, even if FILEWAIT is ON, there cannot be a deadlock that involves any previously explicitly locked object—the server would have already terminated the request.

A deadlock would occur if USERA's program holds a lock on FILEA while waiting for a lock on FILEB. Meanwhile, USERB holds a lock on FILEB while waiting for a lock on FILEA. All the locks are implicit locks, obtained, perhaps, by opening a file for write. USERA is waiting for USERB, while USERB is waiting for USERA. If nothing was done, both USERA and USERB would wait forever.

Or, suppose user USERA's program holds an implicit lock on FILEA while waiting for an explicit lock on FILEB. Meanwhile, user USERB holds an implicit lock on FILEB while waiting for an explicit lock on FILEA, and both USERA and USERB have SET FILEWAIT ON. USERA is waiting for USERB, while USERB is waiting for USERA. This also would cause a deadlock to occur. If nothing was done, both USERA and USERB would wait forever.

The file pool server detects these situations when the deadlock is contained within a single file pool. The server resolves them by rolling back the youngest logical unit of work (the one that started most recently). The server will roll back the logical unit of work even if that user had entered SET FILEWAIT ON.

If you forget what locks you have created on your files, issue the QUERY LOCK command. The QUERY LOCK command also displays the locks created on your file by other users to whom you have granted read or write authority. You can also use the DMSOPDIR (Open Directory) routine with DMSGETDI (Get Directory) or DMSGETDK (Get Directory - Lock).

Using File Pool Space for BFS Files

Before you can create a BFS file in a file pool, the byte file system must be enrolled in that file pool and assigned an allocation of space, thereby creating a BFS file space. To create a BFS file space, you must have permission to connect to the file pool and you must be a file pool administrator.

Directories are in a separate system-owned file space. Therefore, your directory entries do not take up any of the BFS file space. The file pool catalog, within the system-owned space, contains information about the BFS files and directories that exist in the file pool, such as who owns them and who is permitted to look at them.

The file space usage threshold for a BFS file space is always 100% (and cannot be changed), so threshold warnings are not given.

File Pool Restart Recovery

Not all sudden failures of your computer system cause a data loss. When a file pool server is started, it determines whether anyone was making changes when it last ended. That is, the server tries to find unfinished work. If it does, it automatically rolls back any uncommitted changes. By doing this, the server automatically recovers from almost all system failures. However, BFS does not participate in Coordinated Resource Recovery (CRR).

File Pool User Synchronization

The file pool server synchronizes files to allow concurrent access to a file by multiple people. Multiple readers are allowed to a single file, along with one writer. Writers can always see the changes they are making. That is, they can read records they have just written without closing and reopening the file. Readers see changes when they close and reopen the file.

Within the file pool server, there are implicit waits. For example, if a user attempts to open a file for write which is already open for write by another user, and the SET FILEWAIT command was issued with ON, the server waits until the file is closed and committed. If SET FILEWAIT is OFF (the default), there is a rejection of requests instead of waiting on an implicit lock. No waiting ever occurs on a collision with an explicit lock. For example, if a user attempts to open a file for write that has an explicit share lock on it, the request is rejected, regardless of the FILEWAIT setting.

Asynchronous Requests

Asynchronous requests from applications for BFS files and directories are handled in the same manner as for SFS. See “Asynchronous Requests” on page 187.

Chapter 14. Extracting and Replacing System Information

The Extract/Replace facility is a callable service in the Callable Services Library (CSL). The Extract/Replace facility allows application programs to obtain or modify selected system information. The benefit of using Extract/Replace is that it locates the data and returns it to the program without your program being aware of the version of VM, changes in the location of data, or where the data actually resides. The Extract/Replace facility eliminates most CMS control block dependencies.

The Extract/Replace facility can:

- Extract and change data in common control blocks (for example, in NUCON).
- Use search arguments to narrow a search when extracting or replacing data, where multiple instances of that data may exist (for example, information dealing with files, devices, or accessed disks).
- Continue a search through the data looking for the next instance of data that meets the search criteria.
- Change search arguments in the middle of a search loop.
- Return to a specific point in a search loop to extract or replace additional data relating to the original data.
- Set up protected Extract/Replace environments.

This chapter describes:

- How to use the DMSERP CSL routine to extract and replace information.
- An example of a REXX exec using the CSL DMSERP routine. For examples of using the DMSERP routine from high-level language applications, see Appendix E, “PL/I Example,” on page 543, Appendix C, “COBOL Examples,” on page 533, Appendix D, “FORTRAN Examples,” on page 537, Appendix G, “VS Pascal Example,” on page 549, and Appendix B, “C Example,” on page 531.

Using the Extract/Replace Routine

Application programs may need system data that can exist in one place in system storage (single occurrence of data) or it may exist in more than one place (multiple occurrences of data). No matter where or how many places the system information exists, an application can use the CSL Extract/Replace routine, DMSERP, to find the information.

DMSERP has these functions:

- **GETENV**—set up an Extract/Replace environment that is protected against accidental alteration.
- **EXTRACT** and **EXT:***envir*—obtain selected system information
- **REPLACE** and **REP:***envir*—modify selected system information
- **RESET** and **RES:***envir*—initialize an Extract/Replace environment.

For reference information on the DMSERP routine, see the *z/VM: CMS Callable Services Reference*.

Using a Protected Environment

The EXTRACT, REPLACE, and RESET functions of DMSERP operate in the Extract/Replace general environment. All programs have access to this environment and can change data and even reset the environment while another program is using it.

For instance, PROGA could be using tokens to locate information it has found before. If PROGB begins its use of the Extract/Replace facility with a call to the RESET function, all of PROGA’s tokens are lost. To avoid this, PROGA can set up a protected environment by making its first DMSERP call for the GETENV function. GETENV returns a unique environment ID. PROGB cannot issue Extract/Replace requests in this

protected environment unless it has the environment ID. The protected environment lasts until the virtual machine is IPLed or the RES:*envir* function is used.

To set up a protected environment, a program:

1. Uses the GETENV function of DMSERP. DMSERP returns a four-character environment ID (*envir*) in the buffer.
2. Uses the EXT:*envir*, REP:*envir*, and RES:*envir* functions to extract information, replace information, and reset the environment.

Note: The examples in this book of using DMSERP's EXTRACT and REPLACE functions also apply to the EXT:*envir* and REP:*envir* functions in a protected environment.

However, an initial call to RESET cannot be replaced by a call to RES:*envir*. To initialize a protected environment, use GETENV. RES:*envir* reinitializes the environment ID and returns the environment to the system. See "Calling the Extract/Replace Routine from a REXX Program" on page 214 for an example of a REXX program that sets up and uses a protected environment.

Extracting System Information

Use the EXTRACT function of the DMSERP routine to obtain system information. For example, suppose you want to know the storage size of the virtual machine your program will be running in, because your program may not be able to finish normally if the storage size is too small.

If you did not use the Extract/Replace facility, your program would contain code to look in system storage at the location where the virtual storage size information is stored. Problems could occur whenever a new version of VM is used because the storage size information may be in a different location. Your program would then have to be recompiled, or even rewritten, to work on the new version of VM.

If you use the Extract/Replace facility, the portion of program code that looked for the virtual storage size is no longer provided by the application programmer. Your program would only have to call DMSERP and initialize some variables. DMSERP locates the data and returns it to the program while handling any possible changes in the information location. The application program itself does not need to rely on the location of certain data.

Ways of Searching for Data

DMSERP searches for the system information in different ways. You can call DMSERP without any search arguments, using search arguments, using a continued search, or using tokens.

Calling DMSERP without Any Search Arguments

The virtual machine size scenario is an example of a single occurrence of data. Your virtual machine can only be one size at one time; therefore, the storage size information is found in system storage at only one location.

The following REXX exec uses the DMSERP routine, without any search arguments, to extract the storage size of your virtual machine:

```
/* Use DMSERP to extract storage information without
   any search arguments.                                          */

address 'COMMAND'

retcode = 0
size = 0
datatype = 0
len = 4
token = 0

call csl '"DMSERP" retcode "EXTRACT" 0 "VIRTUAL_MEMORY_SIZE"',
         'size datatype len "00000000" "OR" token'

if retcode=0 then do /* Check if Extract/Replace was successful     */

  /* NOTE: The storage size is being returned as a binary           */
```

```
  /* value. This value is being converted to REXX character         */
  /* format using the C2D built-in function.                        */

  size = C2D(left(size,len))%1024
  msize = size%1024
  say 'Storage size = 'size'K ('msize'M).'
end
```

In this example, *retcode, buffer, datatype, buflen,* and *token* are output variables that are filled in by the DMSERP routine. *Numargs* is an input variable. See "Calling the Extract/Replace Routine from a REXX Program" on page 214 for information about converting character data (the C2D function). See the *z/VM: CMS Callable Services Reference* for a description of the parameters of the DMSERP routine.

Calling DMSERP Using Search Arguments

The Extract/Replace routine uses search arguments to search for system information located in more than one place in system storage, such as the status and the access mode of a minidisk.

On each call to DMSERP, you can specify up to ten of these search arguments. DMSERP returns the information matching the search criteria. By combining search arguments, you can define your search to be as broad or narrow as necessary.

See *z/VM: CMS Callable Services Reference* for details on the search arguments of the DMSERP routine.

Example

Here is an example of multiple occurrences of data. Your virtual machine could have several minidisks attached to it. The access mode information for each of these minidisks is found in system storage at individual locations.

Suppose your application wants to store a file it created. Your application can only store the file on a read/write minidisk, and your virtual machine may have access to more than one read/write minidisk. Therefore, your program has two factors to consider:

- Determine the file modes of the accessed disks
- Determine which disks are accessed as read/write.

You can use DMSERP to search for these items. Your application would call DMSERP asking for the file mode and, by passing the appropriate search argument criteria, indicate that the minidisk must be read/write. Extract/Replace starts at the beginning of the minidisk information and keeps searching until it finds a read/write minidisk or it runs out of minidisk information.

The following REXX program calls DMSERP to obtain the access mode of the first accessed read/write disk:

```
/* Extract the access mode of the first read/write disk.          */

address 'COMMAND'
retcode = 0
call csl '"DMSERP" retcode "RESET"'  /* Reset EXTRACT/REPLACE     */

/* Extract the file mode of the first read/write disk.            */


mode = 0     /* INOUT type parameters need to have initial values */
len = 1      /* This is because REPLACE shares parameter definition */
token = 0    /* with EXTRACT and the REXX/CSL interface therefore */
datatype = 0 /* cannot determine the data direction.              */

call csl '"DMSERP" retcode "EXTRACT" 1 "ACCESS_MODE"',
         'mode datatype len "00000000" "OR" token',
         '"CMS_READ_WRITE_DISK" 1 9 1 "EQ"'

if retcode=0 then do /* Check if Extract/Replace was successful  */
  mode = left(mode,len) /* Set access mode in a variable        */
  say 'The access mode of the first read/write disk is' mode'.'
end
```

Calling DMSERP Using Continued Searches

If you want to locate multiple occurrences of data, DMSERP allows your application to continue to search through the system information using the same search arguments previously set in the application without having to go through the entire search process again. A setting on the *flags* parameter of the DMSERP routine sets this continue search feature on.

See *z/VM: CMS Callable Services Reference* for information on the *flags* parameter of the DMSERP routine.

The preceding example locates the access mode of the first read/write disk accessed. By using the continued search setting on the *flags* parameter, your application can easily extract the access modes of all the read/write disks. See "Calling the Extract/Replace Routine from a REXX Program" on page 214 for a REXX program using the continued search feature of the DMSERP routine. Assuming that there is another minidisk that meets the search criteria, the Extract/Replace routine passes back the file mode of the next read/write minidisk for your program to use.

You may also change search arguments while doing a continuous search. For example, you may want your program to store the file on the first read/write minidisk after the disk accessed as A, as long as the minidisk is **not** accessed as C. The initial call to Extract/Replace would contain one search argument indicating a read/write minidisk. After Extract/Replace finds the minidisk with file mode equal to A, you can call Extract/Replace again using the continue search feature and change search argument features. In this second call, there are two search arguments, one indicating the disk must be the read/write and the other indicating that the file mode cannot be C.

Calling DMSERP Using Tokens

DMSERP also uses tokens to search for information. You can think of tokens as addresses (or indexes) telling Extract/Replace to look in a specific place to find information. Tokens are returned to your program every time you extract information where multiple occurrences of that information may exist.

For example, suppose your program has successfully called Extract/Replace and obtained the information pertaining to the file mode of a minidisk. Now, you want to find out how many files are currently on that minidisk. One method is to call Extract/Replace with the exact same search arguments, but this time ask for the number of files instead of the file mode information. This would cause Extract/Replace to go through its entire search argument process again.

However, because Extract/Replace has already found the location of the information concerning the minidisks, your program has already been passed a token representing this location. You need only call Extract/Replace asking for the number of files and passing the token representing the location. A setting of the *flags* parameter of the DMSERP routine signals DMSERP that a token is being passed. Extract/Replace can now locate the information and obtain the information without going through the entire search process again.

See *z/VM: CMS Callable Services Reference* for information on the *flags* parameter of the DMSERP routine.

Changing System Information

You can also use the Extract/Replace routine to change system information. The REPLACE function of the Extract/Replace routine makes use of all the concepts mentioned earlier in this discussion: single and multiple occurrences of data, search arguments, continued searches, changes to search arguments, and tokens. The difference is in the use of the system information. When extracting information, the result of the call to DMSERP provides your application with the system information you requested. When replacing information, your application provides the new information in the call to DMSERP. The result of the call changes the specified system information.

Example 1: This is an example describing a situation when you would use DMSERP with the REPLACE function. Suppose you want to provide the address of a special routine that would be called each time the system reader received a device interrupt. Without using the Extract/Replace routine, you would have to locate the information dealing with the system reader and continually change the address of the special interrupt routine. Because there is information in the system dealing with many devices (disks, tapes, printers), the code needed to find this system reader information could get quite complex.

Using the Extract/Replace routine with the REPLACE function and search arguments, your application can look for information dealing with the reader. You provide the address of the new special routine to replace the old one and Extract/Replace changes the current address with the new address.

Example 2: This is an example of a REXX exec that calls the DMSERP routine with the REPLACE function.

There are two formats of the CMS Ready message—with the time of day displayed or without the time of day displayed. You can use the DMSERP routine with the REPLACE function to set the format of the CMS Ready message.

The following exec, called REPLACE1 EXEC, sets the format of the CMS Ready message so the time of day is not displayed.

```
/* Replaces the time of day setting on the CMS Ready message
   The time of day WILL NOT be displayed after invoking this exec.  */

address 'COMMAND'

retcode = 0
No_Time = '1'    /* Information you want replaced; that is, the     */
                 /* information you input to the DMSERP routine.    */
                 /* This parameter sets the format of the CMS       */
                 /* Ready message. The value '1' indicates that     */
                 /* the time will not be displayed.                 */

datatype = 9     /* Type of data... 9 means 'Indicator'... the      */
                 /* data replaced is either a 1 or 0.               */

len = 1          /* Length, in bytes, of passed data (No_Time)      */

token = 0        /* Used to find or replace additional data         */
                 /* associated with data replaced or extracted      */
                 /* on a prior call. Not used in this example.      */


call csl '"DMSERP" retcode "REPLACE" 0 "NO_TIME"',
         'No_Time datatype len "00000000" "OR" token'

exit retcode
```

After invoking this exec, the time of day setting will not be displayed on the CMS Ready message. The CMS Ready message appears as follows:

```
Ready;
```

The following exec, called REPLACE0 EXEC, sets the format of the CMS Ready message so the time of day is displayed:

```
/* Replaces the time of day setting on the CMS Ready message
   The time of day WILL be displayed after invoking this exec.      */

address 'COMMAND'

retcode = 0
No_Time = '0'    /* Information you want replaced; that is, the     */
                 /* information you input to the DMSERP routine.    */
                 /* This parameter sets the format of the CMS       */
                 /* Ready message. The value '0' indicates that     */
                 /* the time will be displayed.                     */

datatype = 9     /* Type of data... 9 means 'Indicator'... the      */
                 /* data replaced is either a 1 or 0.               */

len = 1          /* Length, in bytes, of passed data (No_Time)      */

token = 0        /* Used to find or replace additional data         */
                 /* associated with data replaced or extracted      */
                 /* on a prior call. Not used in this example.      */


call csl '"DMSERP" retcode "REPLACE" 0 "NO_TIME"',
         'No_Time datatype len "00000000" "OR" token'

exit retcode
```

After this exec has run, the time of day is displayed in the CMS Ready message:

```
Ready; T=0.02/0.03 14:21:29
```

Calling the Extract/Replace Routine from a REXX Program

When calling Extract/Replace from a REXX program, your program needs to convert numeric data where necessary. This is necessary only for Extract/Replace. The C2D and D2C REXX built-in functions convert character format data to numeric format and back. See *z/VM: REXX/VM Reference* for details on the C2D and D2C built-in functions.

A typical program flow is:

1. Call DMSERP routine to get the data you want.
2. Check value in the *datatyp* parameter to determine the data type of the extracted data.
3. If the data type indicates character data or indicator data (bit), the data can be used directly.
4. If the data type indicates numeric data, the data must be converted to REXX character format using `C2D(LEFT(buffer,buflen))`.

   If you are replacing numeric information, the buffer containing the replacement value must be converted from REXX character to numeric using `D2C(buffer,4)`. The *datatyp* parameter must be a fixed binary field with a length of 4 bytes. If you are passing in numeric search values, the buffer containing the search value must be converted from REXX character to numeric using `D2C(sargval,4)`.

This program extracts the number of files on and access modes of all the read/write disks, using a protected environment.

```
/*Report the number of files on and access modes of all R/W disks. */
/*Use a protected Extract/Replace environment (initialized by       */
/*GETENV) instead of the use the general environment (used by the   */
/*EXTRACT, REPLACE, and RESET).                                     */

address 'COMMAND'

/* Initialize variables for parameter list */;
retcode = 0
numargs = 0  /*numargs and infoname parameters are ignored by       */
infoname = 0 /*  GETENV but are required by DMSERP's parameter list.*/
datatype = 0 /* set by DMSERP     */
buffer = 0   /* holds environment ID returned by DMSERP  */
bufflen = 4  /* buffer length must be at least 4 to hold environment ID*/

/* Call DMSERP using GETENV function to initialize a new environment */
/* and receive the environment ID.     */
call csl '"DMSERP" retcode "GETENV" numargs infoname buffer datatype',
         'bufflen'
envir=substr(buffer,1,bufflen) /*get the environment ID from buffer */;

/* Extract the file modes of all disks accessed R/W */
i = 0
flags='00000000' /* Specify initial search    */
mode = 0 /* IN/OUT type parameters need to have initial values*/
number = 0    /* REPLACE shares parameter definition with     */
token = 0     /*  EXTRACT and the REXX/CSL interface, so we   */
datatype = 0  /*  cannot determine the data direction.        */

/* Build the function name to extract from a protected environment */
/* by combining 'EXT:' and the environment ID received from the */
/* GETENV function.       */
function = 'EXT:' || envir

do until retcode ¬= 0
   len = 1
   call csl '"DMSERP" retcode function 1 "ACCESS_MODE"',
    'mode datatype len flags "OR" token',
    '"CMS_READ_WRITE_DISK" 1 9 1 "EQ"';
   flags='10000000' /* Specify continued search  */

   if retcode=0
   then do /* Check if Extract/Replace was successful */
```

```
      mode = left(mode,len) /* Set access mode in a variable  */
      len = 4
      call csl '"DMSERP" retcode function 0 "NUM_FILES"',
       'number datatype len "00100000" "OR" token'
      if retcode=0
      then do
         i=i+1
         accessmode.i = mode
          /* The number of files is returned as a binary value.  */
          /*  Convert this value to REXX character format with  */
          /*  the C2D built-in function.                         */
         filenum.i = C2D(left(number,len))
         say 'There are' filenum.i 'files on the read/write',
            'disk at mode' mode'.'
         end
      end
   accessmode.0 = i
   filenum.0 = i
   end  /* until retcode¬=0   */;

/* Build the function name to restore the protected environment.*/
function = 'RES:' || envir
call csl '"DMSERP" retcode function'

exit
```

Chapter 15. Using Data Spaces

This chapter introduces the concept of data spaces and describes the facilities provided in z/VM that enable you to use them in sophisticated applications. This includes an overview of the CMS callable services library (CSL) routines that simplify the creation, management, and deletion of data spaces.

Introduction

Using ESA/390 extensions to the interpretive-execution facility, z/VM enables applications to take advantage of hardware support for data spaces that are introduced by Enterprise Systems Architecture. An application that runs in an XC virtual machine can create multiple data spaces of up to 2 GB each outside its own primary address space, as shown in Figure 26 on page 217. It can then share these data spaces. This support can be useful for database, three-dimensional graphics, and other applications that require large buffer areas in storage.

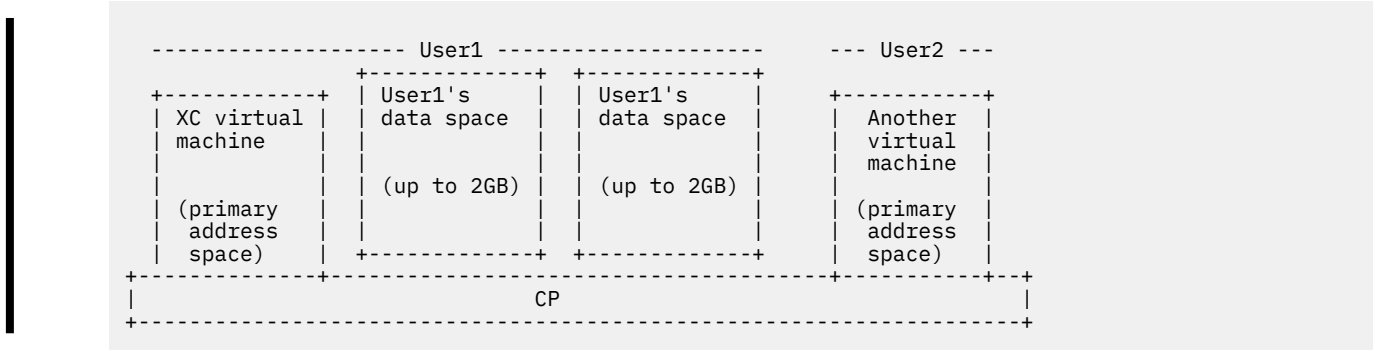


Figure 26. Guest data spaces

Data space support allows an application that executes on an ESA or XA virtual machine to share its primary address space. The application can also copy data from other virtual machines' data spaces into its primary address space.

Terminology

The following sections introduce the terminology and concepts of data space support in z/VM.

Address Spaces

The term address space refers to a contiguous area of storage that can be addressed by a program running in a virtual machine. Every virtual machine has its own ***primary address space*** for programs, data, and control information. A primary address space is the virtual machine's main storage.

Data Spaces

Another type of address space is the data-only address space, which can be requested from the system by an application running in an XC virtual machine. This data-only address space, henceforth called ***data space***, can hold up to 2 GB of data. A data space is an area of storage external to the virtual machine in which the requesting program is executing. A data space can be used only for data storage and manipulation. Programs cannot execute in this area, although a data space can hold programs stored as data. Because a data space has no CMS-reserved areas, it is fully addressable. In addition, access to a data space can be shared with other applications or other users.

Hardware Support—Access Registers

ESA/390 provides a set of ***access registers***, each of which is paired with a general register. The access registers are used to indicate which address space contains the data being referenced by an address in

the associated general register. The access registers are used for addressing only when a program running on an XC virtual machine is executing in ***access-register mode*** (***AR mode***).

The application uses assembler language instructions to control its addressing mode.

Outline of VM Data Space Support

An application that runs on an XC virtual machine can leverage data space support to operate concurrently and efficiently on the following categories of data:

- Data spaces that are external to the application's primary address space
- Data spaces that are external to the primary address space of the application's virtual machine

The following list summarizes VM data space support:

- A data space is a special type of address space (in z/VM publications, data-only address space = data space):
  - It is for data only.
  - The entire space is addressable.
- In XC mode, a data space always resides outside its owner's virtual machine primary address space.
- Data spaces can be shared.
- Any permitted virtual machine within the z/VM system that contains a shared data space can access that shared data space by using CPU instructions.
- Data spaces provide integrity and isolation for the data they contain.
- Only applications that execute in XC virtual machines can reference data spaces directly. However, an application that runs on an ESA, XA, or Z virtual machine can copy data from another user's data space by using z/VM services.
- Access registers (one associated with each general register) allow manipulation of data in a data space.
- The size of a data space can range from 256 pages or blocks (1 MB) to 524,288 pages (2 GB) and is fixed upon creation. If the size specified (in number of pages) is not a multiple of 256, it is rounded up to a multiple of 256.
- CMS provides callable services library (CSL) routines for creating and managing one or more data spaces. (These routines in turn use CP macros to complete the operations.)
- Data in a data space can be manipulated by using most of the ESA/390 assembler language instructions that have a storage-related operand. For more information, see *z/VM: ESA/XC Principles of Operation* and *z/VM: z/Architecture Extended Configuration (z/XC) Principles of Operation*.
- Data spaces can be defined with the following attributes:
  - Scope of program usage
  - An association with CMS events that cause automatic data space-related resource clean up.

  Such a definition contributes to overall VM resource management by assuring the recovery of the data-space related resource.
- XCONFIG directory statements define the scope of a virtual machine's use of data spaces.

Note for MVS Programmers

The principle difference between access-register translation as defined for an ESA/390 virtual machine, like an MVS/ESA guest, and host access-register translation, which is the corresponding process for an XC virtual machine, is that for XC virtual machines, the tables and structures that control architecture usage reside with the host (CP) rather than the guest. Otherwise, XC mode handling of access-register addressing is consistent with ESA/390 mode. The Extract/Replace (DMSERP) callable service may be used to determine if you are executing on an XC virtual machine. If so, then the callable service support to exploit the data space services may be used.

Data Space Support for CMS Virtual Machine Environments

CMS operates in two virtual machine modes, each of which provides a different level of architectural support:

XC

- Can have primary address space of up to 2047 MB
- Can create data spaces up to 2 GB each in size
- Can share data spaces
- Can manipulate data in data spaces
- Can share primary address space

ESA or XA

- Can have primary address space of up to 2047 MB
- Can share primary address space
- Can copy data from a data space using z/VM services.

Uses for Data Spaces

Certain applications require vast amounts of storage to work efficiently. One such class of applications includes databases, which can map an entire data structure into storage at one time rather than overlaying data or explicitly managing I/O. Another class of applications requires large storage buffers, which now can reside outside the virtual machine's primary address space. The graphic representation of three-dimensional objects, image processing, and numerically-intensive computation all require such large storage areas.

How can data space support be exploited by user applications or program products executing on CMS?

Applications in an XC virtual machine can use data spaces to:

- Obtain more storage than is available in its virtual machine's primary address space
- Isolate data from other programs that may execute in the same virtual machine
- Share data located in a data space among programs executing in the same or other virtual machines
- Isolate data by its particular usage and then share that data only among related users (this is an alternative to using a common area that may contain data for various usage)
- Share data located in its virtual machine primary address space with programs executing in other virtual machines.

In addition, applications (particularly those that provide database management services) can use data spaces with the minidisk mapping services. This can improve overall system performance by replacing virtual machine I/O, such as DIAGNOSE I/O, with paging I/O, which is more efficient. See *z/VM: CP Programming Services* for an overview of this support.

Minidisk caching should be turned off for minidisks that are read and written only by MAPMDISK and the corresponding data space. Minidisks that are read and written by a combination of data space access (or MAPMDISK) and virtual I/O (DIAGNOSE and channel program) should be evaluated on an individual basis to determine if minidisk caching should remain enabled.

To reference data in a data space directly, the program must be in AR mode. When in this mode, assembler instructions (such as load, store, and move character) can move data in and out of a data space and manipulate data within it. Assembler instructions can also perform arithmetic operations on the data.

Summary of Data Space Operations

To exploit data space support, use the routines provided in the CMS callable services library (CSL), VMLIB CSLLIB. These routines, which employ CP functions, provide CMS system services for creating, controlling, and deleting data spaces. See the *z/VM: CP Programming Services* for more information on

the CP functions. The application must execute in AR mode for the time necessary to complete any manipulation of or direct reference to data in a data space.

An application running in a virtual machine that is allowed to **own** a data space can perform the following operations, each of which corresponds to a CSL routine:

- Create a data space (DMSSPCC)
- Delete a data space (DMSSPCD)
- Establish addressability to an address space (DMSSPLA)
- Permit other users (virtual machines) or applications to access its address space (DMSSPCP)
- Isolate its address space from other users (DMSSPCI)
- Restore other users' access permission to use an address space (DMSSPCR)
- Release pages of address space storage (DMSSPCRP)
- Remove addressability of an address space (DMSSPLR)
- Ask for the identification token and size of any address space it owns or is permitted to access (DMSSPCQ).

A data space can be created and deleted only from an XC virtual machine that has been authorized using an XCONFIG ADDRSPACE directory statement. With the SHARE option on that directory statement, your application can share address spaces it created with applications running on other virtual machines.

If the application is running in an ESA or XA virtual machine, it must specify BASE for the *name* parameter on the DMSSPCC (Create Data Space) routine. This indicates to the system that data space structures are to be created for the virtual machine primary address space. These structures allow sharing of the primary address space.

An application running in a virtual machine that is allowed to access an address space belonging to another virtual machine can perform the following operations, each of which corresponds to a CSL routine:

- Ask for the identification token of the address space it needs to access
- Establish addressability to the address space
- Copy from the address space into its own virtual storage (only for ESA and XA virtual machines)
- Remove addressability of the address space.

Note: Although data space management also can be performed using CP macros and DIAGNOSE codes, do not use that interface in any program that uses the CMS interface. CMS relies on the CP interface and any mixing of the interfaces may inhibit CMS from performing its error checking. Thus, performing data space management directly using data space CP macros or DIAGNOSE codes along with the CMS interface may cause unexpected results.

Table 25 on page 220 summarizes the data space callable services support in terms of what virtual machine mode the call can be made from and whether the call must be made from the owner virtual machine.

Table 25. Data Space Callable Services Summary

Service :	**May be Called from**		**Is the Owner**	**Not Owner But Permitted (1)**
	ESA or XA mode	**XC mode**		
DMSSPCC - Create Data Space	yes (2)	yes	becomes owner	n/a
DMSSPCD - Delete Data Space	yes (2)	yes	yes	no
DMSSPCRP - Release Address Space Pages	yes (2)	yes	yes	no
DMSSPCQ - Query Address Space	yes	yes	yes	yes

Table 25. Data Space Callable Services Summary (continued)

Service :	May be Called from		Is the Owner	Not Owner But Permitted (1)
	ESA or XA mode	XC mode		
DMSSPCP - Permit Address Space Access	yes (2)	yes	yes	no
DMSSPCI - Isolate Address Space	yes (2)	yes	yes	no
DMSSPCR - Restore Address Space Access	yes (2)	yes	yes	no
DMSSPCPY - Copy from Address Space	yes	no	yes	yes
DMSSPLA - Establish Address Space Addressability	yes	yes	yes (3)	yes
DMSSPLR - Remove Address Space Addressability	yes	yes	yes	yes

Note:

1. The virtual machine that intends to access an address space owned by another virtual machine must be given permission to access by the owning virtual machine and must have the address space identification token (ASIT) or the `name` and `owner` of the data space passed to it (or the name and owner are a fixed for all permitted users) by the owning virtual machine. Having this, the permitted virtual machine must obtain an ALET to be able to access the address space.
2. An application can call this routine only for address spaces owned by the virtual machine in which it is executing. The only address space an ESA or XA virtual machine owns is its primary address space.

   When DMSSPCC is called from an ESA or XA virtual machine, only the name of BASE is accepted. No additional address space is created, but CMS creates and maintains data space structures that allow sharing of the user's primary address space by other virtual machines.

   When DMSSPCD is called from an ESA or XA virtual machine, only the ASIT of the primary address space is accepted. No address space is deleted. However, CMS does delete all data space related structures that were created and maintained for the primary address space.

   When DMSSPCRP is used from an ESA or XA. virtual machine, only zeros can be specified for the ASIT to indicate the primary address space. The pages released are those of the primary address space.
3. The DMSSPLA service is not required to establish addressability to the virtual machine's own primary address space.

Using Data Spaces in Your Applications

This section describes how to use the CSL routines in your applications to set up and manage data spaces.

Creating and Using Data Spaces

A CMS program's ability to create, delete, and access data spaces depends on whether the virtual machine that it executes in has been authorized to do so through CP directory control statements. Because the use of data spaces consumes system resources such as virtual, real, expanded, and auxiliary storage, their use must be controlled. System programmers responsible for tuning and maintaining z/VM use the following directory control statements to control these resources:

- XCONFIG ADDRSPACE directory control statement—This statement authorizes an XC virtual machine to create and delete data spaces, specifies the maximum size of the data spaces, and indicates whether they can be shared. For ESA and XA virtual machines, it allows sharing of the primary address space.
- XCONFIG ACCESSLIST directory control statement—This statement allows an XC, ESA, or XA virtual machine to access more than 62 address spaces (the number allowed without an XCONFIG ACCESSLIST statement), specifying the number of data spaces that can be accessed at any given

time. The access list is maintained for the virtual machine by CP and is used to keep track of address space authorizations. Note that an ESA or XA virtual machine can only reference data in another virtual machine's data space by using the DMSSPCPY (Copy from Address Space) routine.

As an application developer, you should be aware of these installation-established limits and how they relate to return codes associated with the DMSSPCC (Create Data Space) and DMSSPLA (Establish Address Space Addressability) routines.

Creating a Data Space

When a CMS program executing in an XC virtual machine uses the DMSSPCC (Create Data Space) routine to create a data space, it needs to provide the name and size (number of pages) of the data space. In addition, it can specify the attributes to be associated with the data space. Two of these attributes (NOKEEP/KEEP and NOSYS/SYSTEM) determine the life span of the data space. The section entitled "Ownership and Scope of Data Spaces" on page 224 describes what effect each of these options has in CMS.

The NOSHARE/SHARE attribute determines whether the data space can be shared with other users. For more information on sharing, see "Sharing Data Spaces with Other Virtual Machines" on page 226. The storage key and fetch protection attributes (USER/NUCLEUS/OTHER and NOFPROT/FPROT) determine the reliability characteristics of the data space. These attributes are described in the section entitled "Protecting Data Space Storage" on page 231.

With the default attributes, a data space is deleted at end-of-command or abend, no sharing is allowed, the storage key is the same as the user key, and no fetch protection is provided.

When DMSSPCC is called, CMS, utilizing the underlying CP support, returns an ***address space identification token*** (***ASIT***) that uniquely identifies the data space within the scope of a z/VM system IPL (CP IPL). (The ASIT is similar to the STOKEN of MVS/ESA.) The application must retain the ASIT for subsequent data space related service calls. An application can obtain an ASIT of an existing data space by using the DMSSPCQ (Query Address Space) routine, described in "Extracting Address Space Information" on page 229.

An ASIT identifies an **instance** of a data space. An instance of a data space is a temporal concept that means a particular version of the data in a data space. For example, when a data space is deleted and a new data space with the same name is created, the new data space can be considered a new instance of the original data space. Even though the name is the same, recreating a data space results in a new ASIT being assigned. Thus, the ASIT can be used to ensure that only users authorized for a particular instance of a data space can access it.

Accessing Data Space Storage

To gain access to the data space, the program calls the DMSSPLA (Establish Address Space Addressability) routine, specifying the ASIT of the data space, which was returned on the Create or Query call. DMSSPLA adds an entry to the access list and returns an ***access list entry token*** (***ALET***). The access list entry (ALE) identifies the newly created data space and the ALET selects the entry.

The access list referenced in this discussion is associated with the virtual machine's primary address space and is also referred to as the ***host access list***. This access list is equivalent to the primary address space access list (PASN-AL) of MVS/ESA.

Figure 27 on page 223 shows the relationship of an XC virtual machine's address space components. Each CMS user virtual machine has a primary address space where CMS and application programs execute. Associated with a primary address space is the access list that contains entries to data spaces, in this case DataSpaceX and DataSpaceY. When a data space entry is placed on the access list, an ALET is obtained from the system. The ALET indicates to the system which data space to address when making storage references. The application places this ALET in the access register associated with the general register containing an operand whose value is to be fetched from the data space.

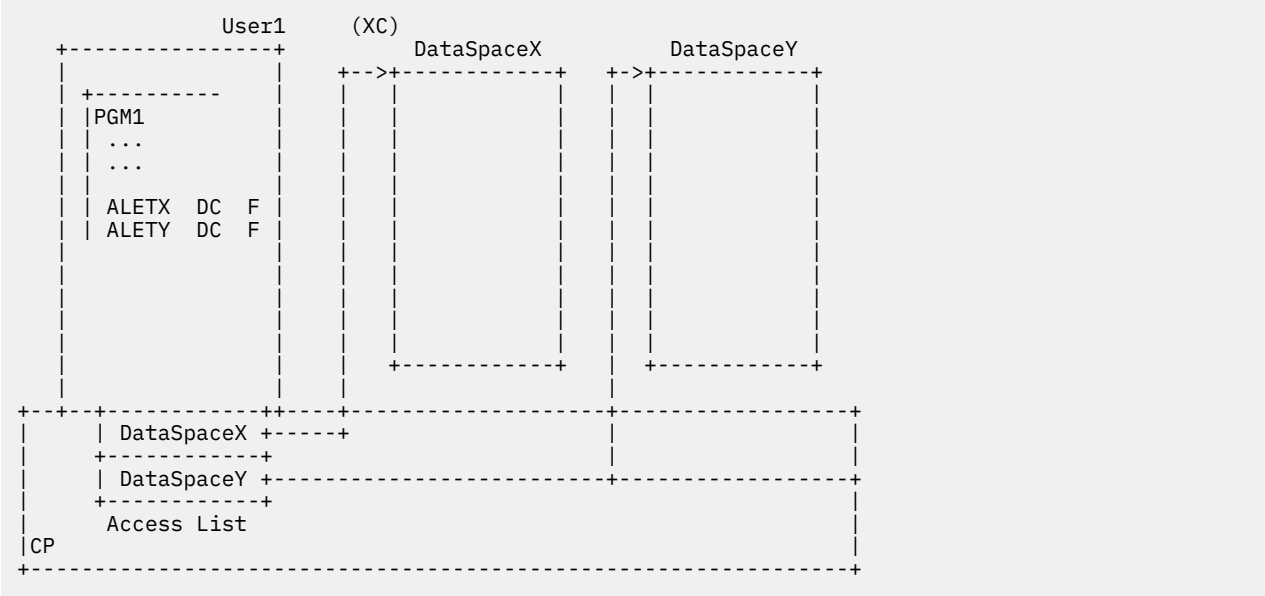


Figure 27. Data Space Addressability

Releasing Data Space Storage

A long-running application, such as a resource manager in a server machine, should release storage when it has finished using the data space for one purpose and wants to reuse it for another purpose, or when it is finished using the data space. Releasing data space storage causes the specified storage page range to be returned to the system. A subsequent reference to the released page will return only zeros. This is a result of the page releasing function.

Releasing data space storage contributes to improving overall system performance and recovery of resources by reducing the ***working set***, the set of pages that must be active to avoid excessive paging. A small working set for a given virtual machine requires less system management and leaves more resources available for system use. Releasing data space storage also results in releasing backing storage for pages that have been paged out. Reviewing the section entitled "Ownership and Scope of Data Spaces" on page 224 will help you determine if your application needs to release data space storage.

To release data space storage, a program running in the owning virtual machine uses the DMSSPCRP (Release Address Space Pages) routine. The ASIT of the data space, the starting page (*offset* parameter) and number of pages (*span* parameter) must be specified on the call. Addressability to the data space must have been established for WRITE and must be in effect when the DMSSPCRP routine is called. The DMSSPCRP routine can also be used to release pages in the virtual machine's primary address space. When doing this, the ASIT value must be specified as 8 bytes of X'00' to designate the primary address space.

Managing Data Space Storage

Managing storage in a data space differs from managing storage in your virtual machine primary address space. Keep the following in mind when you handle your data space storage:

- When you create a data space, request a large enough size to handle the needs of your application. The amount of storage you specify when you create a data space is the maximum size that the data space can be.
- Once you have created the data space and have established addressability to it, the entire addressing range of the data space is available.
- You are responsible for keeping track of how the space is allocated by your application. Depending on the scope of usage attributes defined for the data space when it was created, the system will delete the data space automatically when the related CMS event is encountered. You can delete the data space any time you wish or let the system do it for you once the related CMS event occurs.

- You cannot use CMS storage management services, such as the CMSSTOR, SUBPOOL, DMSFREE, DMSFRET, GETMAIN, and FREEMAIN macros, to manage this area.
- When you are finished using a data space, delete it. The z/VM system will automatically remove the access list entry associated with this data space.

Ownership and Scope of Data Spaces

Events such as end-of-command and virtual machine reset play an important role in the lifespan of data spaces. End-of-command is discussed in more detail in the *z/VM: CMS Application Development Guide for Assembler*. ***Virtual machine reset*** may be caused by using such CP commands as SYSTEM CLEAR, SYSTEM RESET, IPL, or LOGOFF. When virtual machine reset occurs, CP automatically recovers resources associated with the virtual machine.

An application must call a CSL routine to create a data space. From CP's perspective, the data space is owned by the virtual machine in which the application is executing. From a CMS point of view, ownership can be associated with:

- The command cycle in which the creating program was executing: When end-of-command occurs, any data spaces owned by this command cycle are deleted by CMS.
- The virtual machine: Unless the data space is explicitly deleted by a program executing in the virtual machine, CMS does **not** delete the data space.

  Reset of the virtual machine causes any data space created by any program executing in this virtual machine to be deleted by CP.

Scope of Usage within a Virtual Machine

When a program uses the DMSSPCC (Create Data Space) routine to create a data space without specifying any attributes, the data space persists until end-of-command occurs or until it is explicitly deleted prior to end-of-command. A data space of this type, therefore, is accessible to the creating program and potentially to any program it calls directly or indirectly during the command cycle.

For a data space to be accessible, addressability must be established to the data space by calling the DMSSPLA (Establish Address Space Addressability) routine to place an entry for it on the access list. To make the data space accessible to other programs within the virtual machine, the access list entry token (ALET) obtained when establishing addressability must be made available to the other programs requiring access to the data space.

An application can control accessibility to a data space by limiting distribution of the ALET. For example, Figure 28 on page 225 shows how PGM1 creates two data spaces, adds them to the access list, but keeps the DataSpaceX data space private to itself while sharing the DataSpaceY data space with PGM2. In this case, PGM1 uses some related application convention to pass the ALET for the DataSpaceY data space to PGM2.

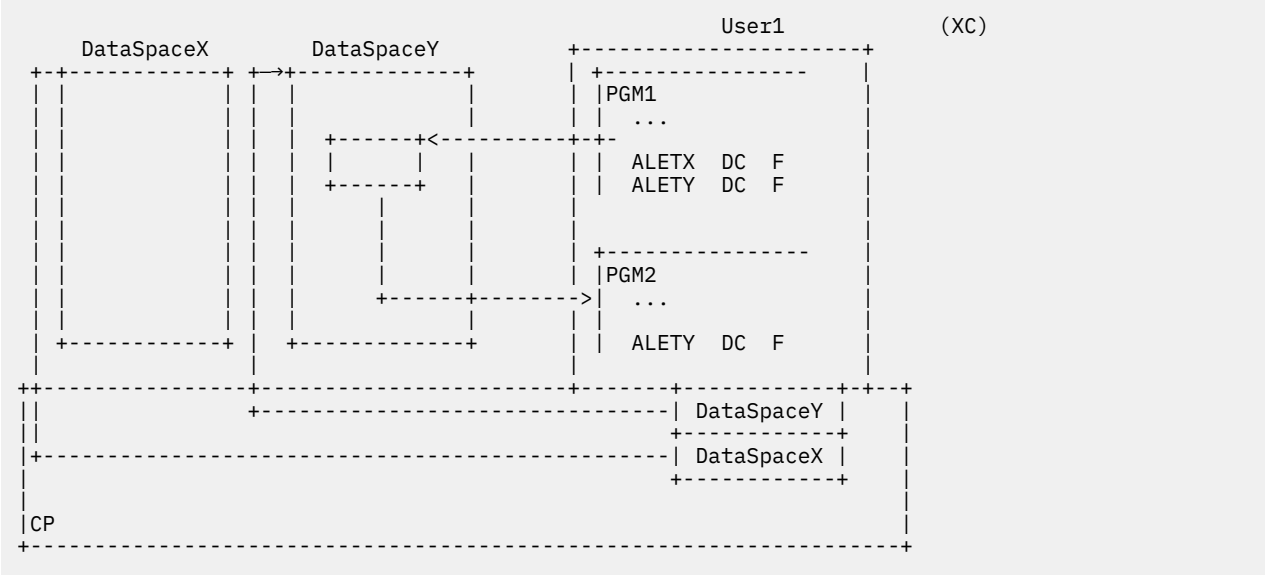


Figure 28. Private and Shared Usage within a Virtual Machine

If an application wants to allow any program executing in the virtual machine, even during different command cycles, to access a data space, it can create the data space with an attribute of KEEP. The KEEP attribute allows a data space to persist through end-of-command. In this scenario, the ALET value must be saved in a known place across program executions. CMS nucleus extension support is suited for this form of application.

Abnormal termination of a program causes CMS abend processing to be invoked. During CMS abend processing, resources not identified to survive abends are recovered by CMS. Specify the SYSTEM attribute on DMSSPCC to enable the data space to survive CMS abend processing. Depending on the application usage, this can be particularly useful when a data space is created with the KEEP attribute. Specifying both SYSTEM and KEEP indicates that the creating application wants the data space to survive should an unrelated application program abnormally terminate in the user's virtual machine.

Recall that data spaces created with or without the KEEP attribute are all viewed by CP as being owned by the virtual machine. In the event of a virtual machine reset, all data spaces owned by the virtual machine are recovered by CP.

Table 26 on page 225 summarizes when a data space is deleted (therefore, the system-related resources are recovered) by the occurrence of CMS events as they relate to the data space attributes.

Table 26. Data Space Cleanup Events

Event	NOKEEP NOSYSTEM	NOKEEP SYSTEM	KEEP NOSYSTEM	KEEP SYSTEM
CMSCALL, SVC 202, OS linkage termination	kept	kept	kept	kept
Command end	deleted	deleted (1)	kept	kept
Abend recovery	deleted	kept (1)	deleted	kept
DMSSPCD (Delete)	deleted	deleted	deleted	deleted
Virtual machine reset, for example, IPL, LOGOFF	deleted	deleted	deleted	deleted

Note: 1. With the NOKEEP and SYSTEM attributes, if an abnormal termination occurs prior to end-of-command, the data space will survive until end-of-command. If no abend recovery is performed using the ESTAE or ABNEXIT exits, the abnormal termination will continue, resulting in end-of-command, at which point the data space would be deleted.

Sharing Data Spaces with Other Virtual Machines

A virtual machine that intends to be the owner of data spaces to be shared with other virtual machines requires a SHARE option on its XCONFIG CP directory statement. A program executing in this XC virtual machine can then create a data space that can be shared with programs executing in other virtual machines. The program creating the data space calls the Create Data Space (DMSSPCC) routine with the SHARE attribute, which tells CMS that this data space is allowed to be shared with other virtual machines.

Table 27 on page 226 summarizes to what degree a data space can be shared based on the attributes assigned to it by its creator.

Table 27. Data Space Sharing Scopes

Data spaces can be shared with:	**NOKEEP NOSHARE**	**NOKEEP SHARE**	**KEEP NOSHARE**	**KEEP SHARE**
All programs that can execute **until end-of-command**	shared	shared	shared	shared
All programs that would execute in the virtual machine, **beyond end-of-command**	not shared	not shared	shared	shared
Programs that would **execute in another virtual machine** (1)	not shared	shared	not shared	shared

Note: 1. The owning virtual machine that intends to share an address space with another virtual machine must be authorized to do so (it must have the SHARE option on its XCONFIG ADDRSPACE CP directory control statement). The virtual machine that intends to access a data space owned by another virtual machine must be given access permission by the owning virtual machine, and must have either the access list identification token (ASIT) or the name and owner of the data space passed to it (or the name and owner is fixed for all permitted users) by the owning virtual machine. In addition, to be able to access the data space, the permitted virtual machine must obtain an ALET by calling the DMSSPLA (Establish Address Space Addressability) routine.

The application that created the shareable data space must authorize the other virtual machines that also need to access the data space. It does this by calling the DMSSPCP (Permit Address Space Access) routine. If the owning virtual machine does not have the SHARE option on its XCONFIG directory statement, the DMSSPCP call fails.

The DMSSPCP routine requires the user ID (or a list of user IDs) or the virtual configuration identification token (VCIT) of the virtual machine that is being authorized and the address space identification token (ASIT) of the data space to be shared. See “Using Alternate User IDs with APPC/VM” on page 232 for more information related to VCIT usage. During the CMS authorization process, the DMSSPCP routine assures that only data spaces created with the SHARE attribute are allowed to be accessible to other user virtual machines.

To obtain access to the data space, an application executing in the authorized user virtual machine must supply the ASIT when calling the DMSSPLA (Establish Addressability) routine. The ALET that is returned on this call is then used by the accessing program when executing in AR mode.

Example

Figure 29 on page 227 is similar to Figure 28 on page 225, except that in Figure 29 on page 227 the DataSpaceY data space is shared between virtual machines. In this case, the Server1 virtual machine owns the data spaces. Application program PGM1 executing in Server1 can pass either the name and owner information or the ASIT of data space DataSpaceY when it receives an APPC/VM request from a related piece of the server application that executes in the user virtual machine. If PGM1 passes the name and owner information of DataSpaceY, PGM2 can call the DMSSPCQ (Query Address Space) routine to obtain the ASIT. If PGM1 passes the ASIT of the data space, then the DMSSPCQ call is not necessary. PGM2 then establishes addressability to the data space by calling the DMSSPLA routine to obtain an ALET that can be used to reference the data space.

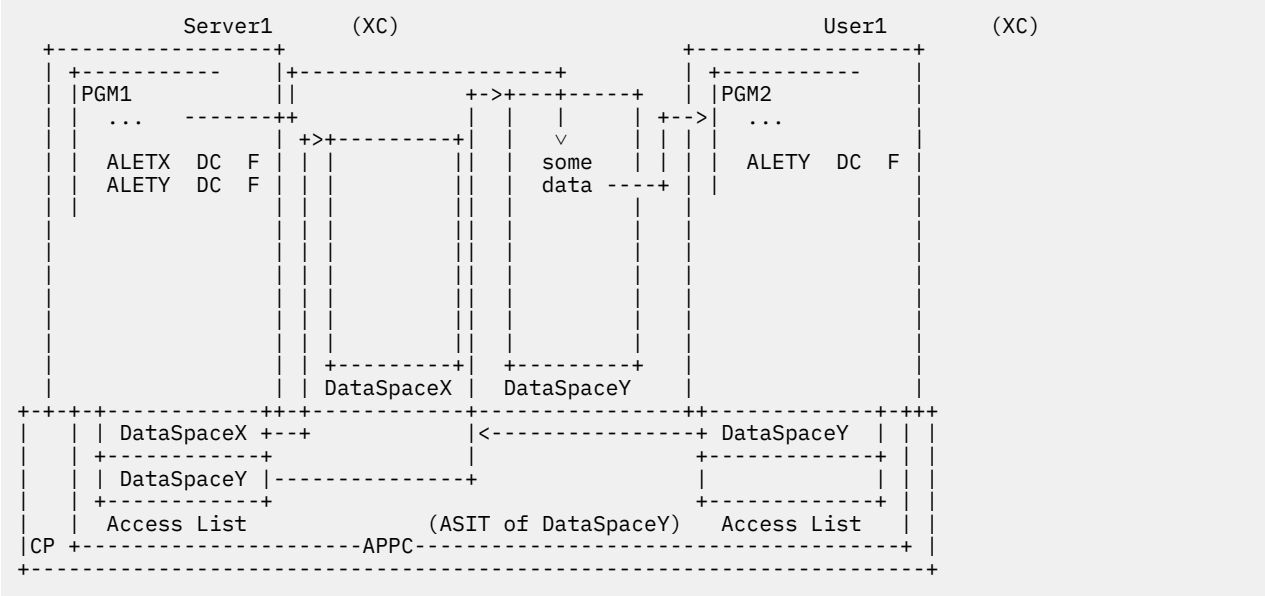


Figure 29. Private and Shared with Another Virtual Machine

Allowing Access to Your Virtual Machine's Primary Address Space

You can allow applications executing in other virtual machines to access your virtual machine's primary address space by performing the same creation and authorization steps as described for data spaces. An application executing in an ESA or XA virtual machine can call the DMSSPCC (Create Data Space) routine as long as the *name* parameter specifies BASE, identifying the primary address space. No data space is created, but CMS establishes and maintains data space structures related to the primary address space. This can be useful when an attribute such as SHARE is used to specify the longevity and sharing intentions for the primary address space.

Next, use the DMSSPCP (Permit Address Space Access) routine to permit access to the primary address space. The entire primary address space will be made accessible to the permitted virtual machines. Note, however, that an XCONFIG ADDRSPACE CP directory statement with the SHARE option is required to allow sharing of the primary address space.

Your program can then pass the ASIT for the primary address space to the program in the permitted virtual machine using APPC/VM. Once it has received the ASIT, the permitted user can call the DMSSPLA routine to establish addressability to your virtual machine's primary address space. Figure 30 on page 228 illustrates this relationship.

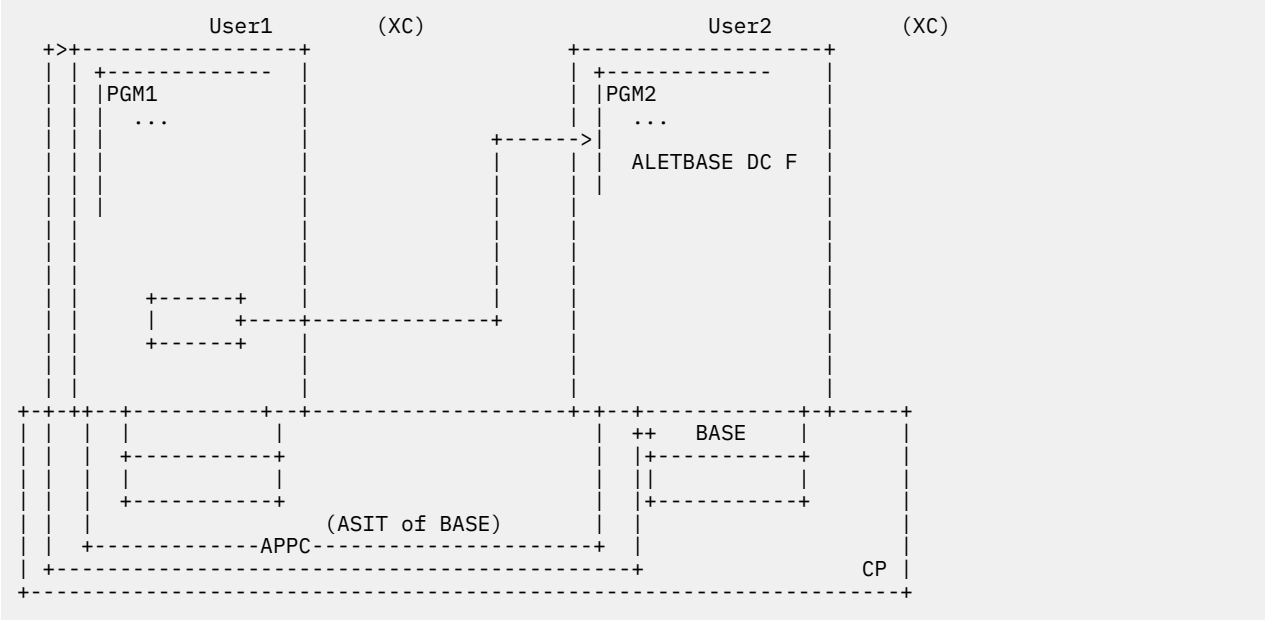


Figure 30. Sharing Primary Address Space with Another Virtual Machine

The User1 virtual machine calls DMSSPCC, specifying BASE for the *name* parameter, to make its primary address space shareable. (This assumes an XCONFIG ADDRSPACE statement with the SHARE option in User1's CP directory.) It then calls DMSSPCP to permit User2 and passes the ASIT returned on DMSSPCC to the User2 virtual machine. The PGM2 program in the User2 virtual machine calls DMSSPLA with the ASIT to establish addressability to the User1 primary address space. The DMSSPLA call places the ALET for the User1 primary address space on User2's access list. This allows PGM2 in User2 to access User1's primary address space. When PGM1 in User1 calls the Permit routine, it can specify whether READ or WRITE access is granted.

When your virtual machine's primary address space is shared with other users, SEGMENT LOAD and SEGMENT RELEASE commands cannot be issued from your virtual machine. Sharing your primary address space does not affect existing segments, but you cannot release those segments or attach new ones.

Isolating Shared Address Spaces

An application running in a server virtual machine, commonly referred to in z/VM as a ***server application***, may need to modify a data space that is shared with a number of user virtual machines. In some cases, the application must make the modification in a **private** state, when the application can be assured that there are no user references in process to the data space. The DMSSPCI (Isolate Address Space) routine provides this function, where upon return from the service, the data space is in a private state.

Once the server application completes the modifications to the data space, it can call the DMSSPCR (Restore Address Space Access) routine to re-permit users that were permitted access to the data space before the DMSSPCI call. The restore call re-permits users that are still logged-on (even if they are disconnected) at the time of the call. These users, however, must re-establish addressability to the data space before they are allowed to access the data again, so they will need to be able to detect when the address space has been isolated by its owner for a refresh or update.

If a reference is made to the data space after the isolate was performed by the owner, the user virtual machine receives an ***addressing-capability*** exception. The user application can use this exception to detect the isolate condition and then can call the DMSSPLA routine to re-establish addressability to the data space and obtain a new ALET for referencing the data space. If the DMSSPLA routine fails with a return code indicating that the restore has not yet been performed, the application must have some mechanism to either retry or inform the user that the function cannot be performed at this time. Thus, the isolate/restore support therefore involves more complicated support in the portion of the server application that executes in the user virtual machine.

Note: An application requiring a shared data space that periodically needs to be isolated for refresh activity can utilize its own locking protocol that may be less costly in terms of overall system performance.

Extracting Address Space Information

An application can obtain information about an existing data space owned either by the virtual machine in which it is executing or by another virtual machine. The DMSSPCQ (Query Address Space) routine returns the ASIT and size of the data space whose name and owner are specified on the call. The owner is defined to be the user ID of the virtual machine from which the data space was created. Specifying an asterisk (*) for the *owner* parameter indicates that the owner of the data space is the virtual machine in which the program is executing.

To obtain information about a data space created from another virtual machine, the virtual machine in which the program is executing must already have been permitted access by the owning virtual machine. In this case, the requesting program must also specify the user ID of the owner of the data space on the DMSSPCQ routine call. See "Sharing Data Spaces with Other Virtual Machines" on page 226 for more information.

Using the DMSSPCQ routine, an application developed to use shared data spaces can determine the size of the data space so it can ensure that references will not produce addressing exceptions. Note, however, that when used to query the size of a primary address space, DMSSPCQ returns the highest address accessible in the virtual machine. When discontiguous shared segments are being used, it is possible that gaps may exist between a saved segment and the primary storage of the virtual machine. Therefore, addressing exceptions may still be possible in this situation.

Rules for Creating, Deleting, and Using Data Spaces

To protect data spaces from unauthorized use, the z/VM system (CP and CMS) uses certain rules to determine whether a program can create or delete a data space or whether it can access data in a data space. The rules for CMS programs are similar to MVS/ESA supervisor state programs, because all programs invoked by CMS execute in supervisor state.

The following rules apply to CMS programs using the CSL routines for data space support.

Rules for Creating Data Spaces

Only a program running in an XC virtual machine can create a data space that will reside outside the virtual machine's primary address space. A program running in an ESA or XA virtual machine can specify BASE on the Create Data Space routine to share its primary address space. The virtual machine must be authorized with a XCONFIG ADDRSPACE CP directory control statement.

Rules for Deleting Data Spaces

A program can delete a data space only if it is running in the virtual machine that created it.

Rules for Releasing Storage in Address Spaces

A program can release an area of an address space only if the address space was created from the same virtual machine.

Rules for Establishing Addressability to Address Spaces

Establishing addressability involves specifying the ASIT of the address space when calling the DMSSPLA routine. This routine places an entry on the access list and returns an ALET that designates the entry.

A program can establish addressability to any data space owned by the virtual machine in which it is executing. If the data space was created from another virtual machine, access authority must first be granted by the owning virtual machine. The owning virtual machine grants access by calling the DMSSPCP (Permit Address Space Access) routine. The owning virtual machine must also be authorized to share its address spaces through the SHARE option on the XCONFIG ADDRSPACE CP directory control statement.

If an application needs to access more than 62 data spaces, an XCONFIG ACCESSLIST CP directory control statement is also required in the virtual machine's CP directory to give the virtual machine a bigger access list.

Applications running in ESA or XA virtual machines can access a data space created by an XC virtual machine by obtaining the ASIT of the data space and then specifying it on a call to the DMSSPLA routine to establish addressability to the data space. The ALET obtained from this call must be saved for subsequent use with the DMSSPCPY (Copy from Address Space) routine. Figure 31 on page 230 illustrates this environment, in which PGM1 creates data space DataSpaceY and stores some data in it. It then permits the User2 virtual machine access and uses APPC/VM to pass the ASIT for the data space to PGM2. PGM2 in turn calls the Establish Addressability routine and then the Copy from Address Space to copy data from DataSpaceY to its own primary address space.

Rules for Accessing Data in Address Spaces

Once it has established addressability to the address space, a program executing in AR mode can access the address space. Note that address space storage is also subject to storage key protection and ***access-list-controlled*** protection. Access-list-controlled protection means that read and write access to an address space is regulated by the CP-controlled access list associated with the virtual machine.

Applications running in ESA or XA virtual machines can only reference data spaces by calling the DMSSPCPY routine, which copies data from the data space to their primary address space.

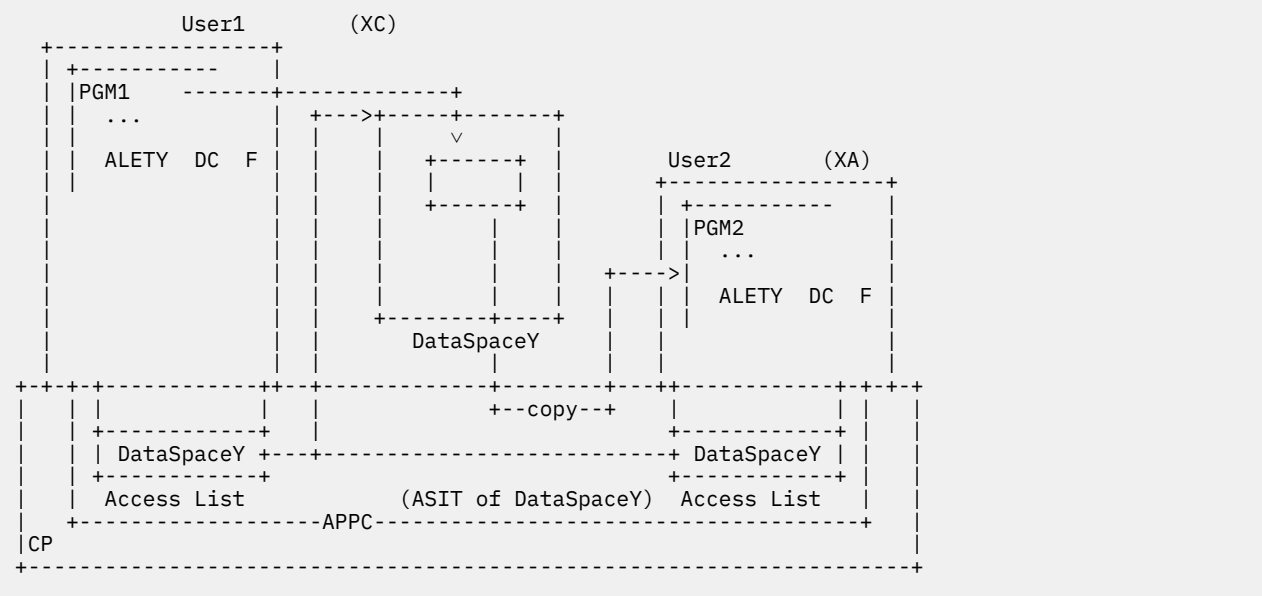


Figure 31. Sharing with a XA-Mode Virtual Machine

Using VM Data Space Services from ESA or XA Virtual Machines

Although applications cannot create data spaces from ESA or XA virtual machines, the data space callable services can be used to share the virtual machine's primary address space with other virtual machines.

This may be useful when a server application that executes on an XC virtual machine requires services from another server that may be capable of running only on an ESA or XA virtual machine. In this case, the non-XC server application cannot directly access a data space that could be used to pass information between servers. The non-XC server application, however, can establish a communication area within the primary address space of the ESA or XA virtual machine on which it runs. The location of this area can be conveyed to the XC server application, which can then directly reference the specified area. The non-XC server application can store/fetch to the designated area because it is within its primary address space. The XC server application can also store/fetch to this area because it has the extended addressing capabilities of ESA/390 architecture. Of course, the non-XC server application would have to follow the same requirements for sharing the primary address space as for sharing a data space. See the

section entitled “Allowing Access to Your Virtual Machine's Primary Address Space” on page 227 for more information on sharing the primary address space.

Protecting Data Space Storage

The means used to protect data space storage depends on the sharing environment the application is working in. We will discuss sharing within a virtual machine first, then turn our attention to sharing with other virtual machines.

Sharing within a Virtual Machine

The owning program of a data space that is shared with other programs executing in the **same** virtual machine can use the access list entries to control access to a data space by calling the DMSSPLA routine. It can do this by obtaining two ALETs:

- One with the WRITE option, for its own use
- One with the READ only option, which it would pass to the programs that required access to the data space.

This technique is known as access-list-controlled protection because it uses the entries in the access list to control read and write access to a data space.

Example

Figure 32 on page 231 illustrates this form of protection. PGM1 creates data space DataSpaceX and then calls DMSSPLA twice to obtain two ALETs: One ALET for read/write to use for its own needs, and the other ALET for read-only to pass on to PGM2. In this case, PGM2 can only read data in data space DataSpaceX; any update attempt will result in a protection exception.

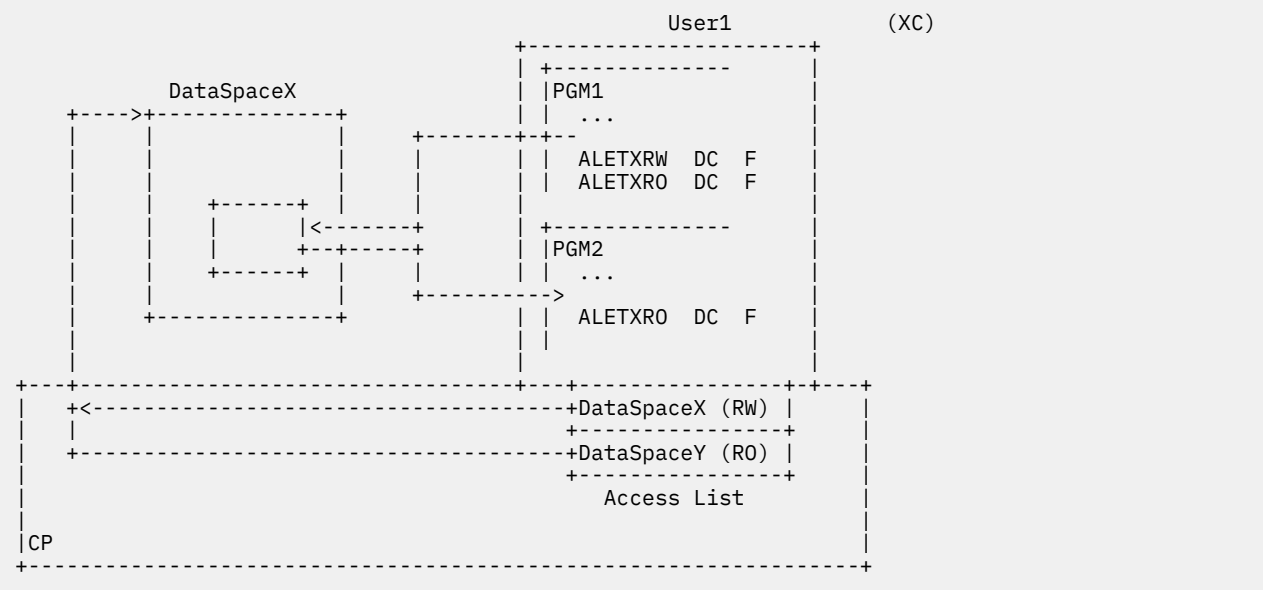


Figure 32. Access-List-Controlled Protection

Sharing With Other Virtual Machines

When sharing a data space with other users, more storage protection options are available to the owner of the data space. Access-list-controlled protection can still be used by calling the Permit Address Space Access (DMSSPCP) routine **without** specifying the WRITE option, thus taking the default READ option. This allows only READ access when the permitted user calls the DMSSPLA routine to establish addressability to the data space. Because there is no way for another user to modify the ALET type from read to write, or to access the data space without the ALET, this is a reliable means of protecting data space storage.

A creating program can also protect a data space by specifying the FPROT and KEY parameters on the DMSSPCC routine. KEY assigns the storage key for the data space and FPROT specifies that the storage in the data space is to be fetch-protected. Storage protection and fetch protection apply to the entire data space. For example, a program cannot reference storage in a fetch-protected data space without executing in the PSW key that matches the storage key of the data space or PSW key 0. Remember that user programs in CMS execute in virtual machine supervisor state, so they can change the PSW key at will. Therefore, storage key and fetch protection may not provide adequate security in all cases.

Use the NUCLEUS and USER attributes on the DMSSPCC routine to ensure a storage key consistent with the PSW key of your application. You can also specify the fetch-protection (FPROT) option with either USER or NUCLEUS when creating the data space.

Other Considerations When Using VM Data Spaces

The following sections address other important issues that you may need to consider when writing an application that uses the data space support.

Using Alternate User IDs with APPC/VM

To authorize another virtual machine to access a data space, the application in the owning virtual machine calls the DMSSPCP (Permit Address Space Access) routine, specifying the logon ID (user ID) of the virtual machine to be permitted access. In cases where the owner of the data space is a server (for example, Shared File System (SFS)), the server usually obtains this user ID from information that is supplied by APPC when the APPC connection is established between the user and the server.

When a worker virtual machine (one that does work for other user virtual machines) is involved, however, there is an additional consideration, because worker virtual machines can run with an alternate user ID in effect. The alternate user ID is the user ID of the virtual machine on whose behalf the worker machine is performing the task. When the worker using an alternate user ID connects with the resource owner, APPC/VM reports the alternate user ID, rather than the logon ID, as the identity of the virtual machine making the connection.

Depending on how the application is written, a worker virtual machine might require access to a data space to perform work on behalf of the requester (user). In this case, the worker's identity needs to be known because it is the worker virtual machine that needs to access the resource on behalf of the user. The ***virtual configuration identification token*** (***VCIT***) provides the identity of the APPC/VM connecting virtual machine, in this case the worker virtual machine. Thus, when a worker virtual machine is to be the permitted user of a data space, the resource owner virtual machine would specify the VCIT for the *user* parameter along with the VCIT option on the DMSSPCP routine, rather than specifying the alternate user ID.

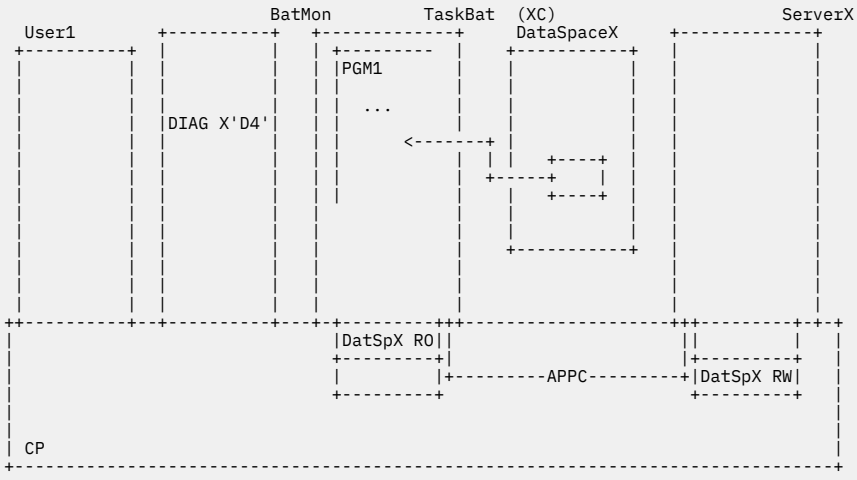
A virtual machine using APPC/VM to communicate will have the VCIT of the connecting virtual machine passed as part of the connection pending extended data provided on an APPC connection request. When an APPC connection flows through TSAF, AVS, or an ISFC line (that is, when the connection originates outside of the system containing the target virtual machine), the VCIT field is zeros.

Note: Virtual Machine Communication Facility (VMCF) and Inter-User Communication Vehicle (IUCV) communication always supply the logon ID and never an alternate user ID or secure VCIT on their connection requests. Therefore, data space access permission must be granted using the user ID of the virtual machine making the connection (for example, a worker virtual machine) when using either of these means of communication.

Example

Figure 33 on page 233 shows a user virtual machine using APPC/VM to submit work to a batch-monitor virtual machine (BatMon). The batch monitor in turn assigns a task virtual machine (TaskBat) to do the work on the user's behalf. However, the TaskBat virtual machine needs a resource owned by ServerX, for which the user virtual machine is authorized. The batch monitor virtual machine issues a DIAGNOSE code X'D4' to change TaskBat's alternate user ID to that of the user ID on whose behalf the work is being done. Then TaskBat issues an APPC/VM connection to ServerX to get at the resource required. Both the alternate user ID and the VCIT are passed on the connect. The ServerX virtual machine uses the alternate

user ID on the pending connect request to validate that the user is authorized for the resource. If the user is authorized, the VCIT on the pending connect request is used to allow access to data space DataSpaceX by TaskBat. After TaskBat gets a connect accepted and the ASIT of the data space is reflected back to it, it can call the DMSSPLA routine to establish addressability to the data space.



1. Batch monitor issues DIAGNOSE code X'D4'
2. CP passes User1's user ID and VCIT of TaskBat on connect of TaskBat to ServerX

Figure 33. Alternate User ID and VCIT Usage

Storage Error Notification for Access Register-Specified References

An application can choose to be notified when a storage error occurs 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 interrupt is reflected to the XC virtual machine. This type of error is typically caused by a:

- Real error in 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'.

The application program can establish either an ESTAE or CMS ABNEXIT routine to handle these possible abend occurrences. In the case of the X'1F4' abend code, CMS releases the page identified by the "failing storage address" information before driving any ABNEXIT or ESTAE exit routine. This action is indicated to an ABNEXIT routine through the SDWFSPRL flag in the SDWFLAG2 field of the CMSSDWA DSECT. If CMS could not release the affected page, this flag will be off.

In the case of the X'1F5' code, the exit routine can still save the data from the affected page. The X'1F5' abend is primarily associated with a mapped page. The exit routine should **unmap** the page identified by the failing storage address or map the page to a different DASD block. For more information on recovery actions, see *z/VM: CP Programming Services*.

Note:

1. If the exit routine attempts to recover from the abend without returning to CMS and a X'1F4' abend was indicated, the exit should ensure that the affected page was released. The exit can call the DMSSPCRP routine or issue a DIAGNOSE code X'10' to release the affected page if CMS has not already done so.
2. In a z/XC virtual machine, the failing storage address is eight bytes. However, if the high-order four bytes of that address are zero, CMS provides a four-byte address that is compatible with the ESA/XC architecture. For applications that do not support more than two gigabytes, and in particular for I/O errors that are associated with VM data spaces, this means that no changes are required to handle z/Architecture eight-byte failing storage addresses.

The IHASDWA mapping macro is used by an ESTAE exit while the ABNEXIT exit uses the DMSSDWA mapping macro to determine the address/data space that was involved with the error as well as the page affected.

Virtual Machine Event Handler

Data space support introduces new virtual machine notification events. These events relate to the use of function that produces asynchronous external interrupts in the form of a X'2603' interrupt code to an XC virtual machine. When CMS detects a X'2603' interrupt, it invokes an application-specified handler routine to process the notification of the event. An application must use HNDEXT SET to define a X'2603' external interrupt handler to CMS. A X'2603' external interrupt handler must be defined by an application that intends to use the SAVE function of the CP MAPMDISK macro or ***page-fault*** notification provided by the CP PFAULT macro. These macros are described in the *z/VM: CP Programming Services*.

Page-Fault Notification for Access-Register-Specified References

An application can choose to be notified when a page fault occurs 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.

Note: The term page fault as used throughout this chapter also includes ***segment faults***. Segment-fault notification is reflected through the same mechanism as page faults.

Page-fault notification allows the application to avoid page-fault serialization by overlapping CP's page-fault resolution for one application task with the execution of a different application task.

The page-fault notification discussed below is **not** related to the support provided by the CP SET PAGEX command. Page-fault notification is not affected by SET PAGEX in any way.

For each valid access list entry, page-fault notification can be requested for storage references made using the access list entry. An application requiring this support must utilize CP and CMS services. To activate and be able to use page-fault notification, the application must:

1. Establish an external interrupt handler by invoking the CMS macro HNDEXT
2. Issue the CP PFAULT macro to define a TOKEN address to CP
3. Call the DMSSPLA routine with the ASYNC option when establishing addressability to the data space.

When page-fault notification is in effect, an application can control the enabling and disabling of the interrupt by using the CMS ENABLE macro interface to selectively control the external interrupt mask in the PSW. An application can cancel page-fault notification using the CP PFAULT CANCEL function.

When a page fault occurs on an application reference to a data space and page-fault notification enablement is in effect, a handler routine defined by the HNDEXT macro SET function is driven. See the *z/VM: CMS Application Development Guide for Assembler* for a description of this support and operation.

Overview of CMS Service Call Support in AR mode

To reference data spaces directly, CMS programs execute in an XC virtual machine in access-register mode (AR mode). The application program enters AR mode by issuing the **Set Address Space Control** (**SAC**) instruction:

```
SAC 512
```

The results of this instruction cause bits 16 and 17 of the PSW to be set to a B'01' value. Subsequent instructions continue to be fetched from the **primary** address space of the virtual machine. The instruction operand addresses, however, may refer to an address space other than the primary address space of the user virtual machine. The access register that is associated with a base register of an instruction is now used to determine the ***operand address space***. See "AR Mode Execution Considerations" on page 239 for a programming example.

As a general programming practice when using AR mode, the access registers associated with the general registers used as base and save registers should always contain a value of 0 to indicate the primary address space.

An application executing in AR mode can use CMS services by calling CMS-supported native and simulated interfaces. The various CMS application programming interface groups provide support as described in the following sections.

Effect of Data Space Support on Preferred Programming Interfaces

The preferred programming interface comprises CMS preferred macros and CSL routines. This section describes the support provided by these two types of preferred interface calls.

Preferred CMS Macros

Macros from this group can be called while in AR mode. In general, calls to preferred interface group macros are made by supervisor assisted linkage (SVC) transfer to the called service. When an application calls a preferred-group macro, control is transferred to the macro in primary-space mode. Upon completion of the macro, control is returned to the caller in the addressing mode in effect when the call was made.

To prevent inadvertent modification of a data space (or other address space) by the preferred group macro expansion code while executing in AR mode, ensure that a `DMSSTATE SET,ASCENV=ARM` macro specification is in effect before calling a preferred macro. This will assure that the caller's access register 1 is reset to 0, ensuring that references made in AR mode using general register 1 refer to the caller's primary address space.

The DMSSTATE macro, similar to the MVS/ESA SYSSTATE macro, is provided to condition an assembler global variable that will cause AR mode toleration code to be expanded within the CMS preferred group macros during program assembly. If the global variable indicates an AR mode environment, the additional expansion takes place; otherwise, it does not. If a new AR mode application program contains a `DMSSTATE SET,ASCENV=ARM` coded before any CMS preferred group macros are called, any AR mode environment code contained in the macros will be expanded. If the application uses only CMS preferred group macros, it can issue the SAC 512 instruction and remain in AR mode throughout its processing.

If a preferred macro is called while in AR mode, but a prior `DMSSTATE SET,ASCENV=ARM` macro specification is not in effect and access register 1 contains a nonzero value, the call is terminated by CMS with a X'1CD' abend code. The X'1CD' abend indicates that CMS detected a possible inadvertent modification of a data space during the preferred macro expansion.

If access register 1 contains a valid ALET value in AR mode, the preferred macro may produce a code expansion that will attempt to modify the address space associated with the ALET. This may result in a program check or an inadvertent space modification. If a program check does not occur during the macro code execution, CMS subsequently terminates the call with a X'1CD' abend code.

The preferred group macros are:

- Downward compatible with CMS releases that introduced the macro to the preferred group
- Compatible with assembler F and non-ESA assembler H
- Available in all virtual machine modes.

The CMS services called through this interface are performed in primary-space mode (non-AR mode). All parameters specified as storage addresses are considered to refer to the primary address space of the

user's virtual machine (for example, the PLIST must reside in the primary address space). Similarly, any save areas required must also be in the primary address space.

Access registers are saved across the interface call and are restored upon return to the caller. The caller's translation mode (primary space mode or AR mode) is restored to what it was at the time of the call.

SVC Entered Services
The CMS supervisor indicates to a callee, that is, the target of a CMSCALL macro, whether the caller was in AR mode at the time of the call. The USEAR flag in the USEMFLG field of the USERSAVE mapping macro indicates whether the call was made in AR mode. When the callee gets control, it will be in primary space mode and register 13 will point to USERSAVE.

BRANCH Entered
Macros for services that have documented branch entry interfaces save and restore the access registers and the translation mode across the interface call. Once entered, the function is performed in primary space mode.

Note that access registers 0, 1, 14, and 15 are volatile across macro calls.

Existing programs containing CMS preferred macros must be reassembled in order to run in AR mode. Reassembling picks up the new version of the macro interface. If the programs are not reassembled and are issued in AR mode, the programs abends with a CMS abend code of X'1CD'. The abend occurs because a nonzero value is detected in the access register associated with the parameter list pointer. An abend code of X'1CD' indicates this.

Interrupt Handling in AR Mode

z/VM saves the access registers across all CMS interrupt handling, whether in AR mode or primary space mode in an XC virtual machine. The access registers and translation mode (primary space mode or AR mode) are restored when the interrupted process resumes.

Exit routines (for example, HNDEXT, HNDIO) are driven in primary space mode from the first level interrupt handler (FLIH) in an XC virtual machine. At entry to the interrupt handler, access registers are the same as they were at the time of the interrupt. Control should be returned to the FLIH in primary space mode.

Callable Services Library Routines

Before attempting to call CSL routines from a high-level language, check the reference manual for the language to determine whether such calls are supported and what setup operations are required.

Assembler language programs can call CSL routines from AR mode using either the MVS/ESA level of the CALL macro or using the CMS preferred interface group "fast path" macro CSLFPI.

The CMS services called through these interfaces are performed in primary space mode. All parameters specified as storage addresses are considered to refer to the primary address space of the user's virtual machine.

Using the CALL Macro with the CMSCSL Interface

The MVS/ESA CALL macro can be used to invoke a CSL routine through the DMSCSL interface. This CALL macro must be conditioned at assembly time. Conditioning is performed in MVS/ESA using the SYSSTATE macro with the ASCENV=AR parameter specified to indicate an AR mode call. This macro call ensures that the parameter list (PLIST) modification of the CALL macro expansion does not inadvertently modify a data space or the primary address space of another virtual machine. The MVS/ESA levels of the SYSSTATE and CALL macros are provided by CMS.

The callable service routine called by the interface operates in primary space mode. All parameters specified as storage addresses are considered to refer to the primary address space of your virtual machine.

If a service call is made using a non-MVS/ESA level of the CALL macro or made with the MVS/ESA level of the CALL macro but not conditioned by SYSSTATE ASCENV=AR, then the caller must ensure that access register 1 has a value of 0. If a nonzero value is detected in the access register associated with the

parameter list pointer, the call will be terminated with an abend code of X'1CD'. This test is common to all language calls. Parameters specified as storage addresses must refer to the primary address space of the user's virtual machine, otherwise unpredictable results may occur.

When the CALL macro is executed, access register 13 should have a value of 0 to avoid the inadvertent modification of an address space or the occurrence of a program check.

Using the Fast Path to Invoke CSL Routines

In CMS, specifying TYPE=CALL on the CSLFPI macro causes a direct transfer to the called service. When an application calls a CSL routine while in AR mode, control must be transferred to the routine in primary space mode. Specifying TYPE=CALL on the CSLFPI macro with a prior `DMSSTATE SET,ASCENV=ARM` macro specification in effect ensures that this condition is met. Upon completion of the routine, control is returned to the caller in the addressing mode in effect when the call was made.

Do not use the CSLFPI macro in AR mode unless you also use the `DMSSTATE SET,ASCENV=ARM` macro specification. If a call to the CSLFPI TYPE=CALL macro is made in AR mode without the conditioning DMSSTATE macro specification and access register 1 contains a valid ALET value, the call may produce a code expansion that will attempt to modify the address space associated with the ALET. This can result in an inadvertent modification to an address space or in a program check.

The DMSSTATE call prevents the CSLFPI TYPE=CALL macro expansion code from inadvertently modifying an address space while executing in AR mode by resetting the caller's access register 1 to 0, ensuring that references made using general register 1 refer to the caller's primary address space.

To save the translation mode (AR mode or primary space mode) and access registers across a call while executing in AR mode, ensure that a `DMSSTATE SET,ASCENV=ARM` macro specification is in effect whenever your application calls CSLFPI with TYPE=CALL, TYPE=AREA, or TYPE=DSECT specified.

When the CSLFPI TYPE=CALL macro is issued, access register 13 should have a value of 0 to avoid the inadvertent modification of an address space or the occurrence of a program check.

Table 28 on page 237 shows the results of using the CSLFPI and DMSSTATE macros to call a CSL routine while in primary space mode. The letter "P" in the fifth column represents primary space mode.

Table 28. CSLFPI TYPE=CALL Behavior on XC Virtual Machine in Primary Space Mode

Assembly time definition		**Results of CSLFPI call**			
Before Issuing CSLFPI TYPE=AREA, Call DMSSTATE SET,ASCENV=	**Before Issuing CSLFPI TYPE=CALL, Call DMSSTATE SET,ASCENV=**	**Caller's AR1 Reset to 0**	**Caller's Trans. Mode & ARs Are Saved**	**Callee Gets Control in**	**Caller's Trans. Mode & ARs Are Restored**
NOARM	NOARM	no	no	P	no
NOARM	ARM	yes	no	P	no
ARM	NOARM	no	no	P	no
ARM	ARM	yes	yes	P	yes

Table 29 on page 238 shows the results of using the CSLFPI and DMSSTATE macros to call a CSL routine while in AR mode. The letter "P" in the fifth column represents primary space mode, while the letters "AR" represent AR mode.

Table 29. CSLFPI TYPE=CALL Behavior on XC Virtual Machine in AR Mode

Assembly Time Definition		Results of CSLFPI Call			
Before Issuing CSLFPI TYPE=AREA, Call DMSSTATE SET,ASCENV=	**Before Issuing CSLFPI TYPE=CALL, Call DMSSTATE SET,ASCENV=**	**Caller's AR1 Reset to 0**	**Caller's Trans. Mode & ARs Are Saved**	**Callee Gets Control in**	**Caller's Trans. Mode & ARs Are Restored**
NOARM	NOARM	no	no	AR	no
NOARM	ARM	yes	no	P	no
ARM	NOARM	no	no	AR	no
ARM	ARM	yes	yes	P	yes

Effect of Data Space Support on Compatibility Programming Interface

Calling macros from this interface group in AR mode results in an abend with a CMS abend code of X'1CD'. Access registers as well as the general registers are preserved on entry, and are made available to an ABNEXIT or ESTAE routine.

The CMS supervisor detects the execution of a compatibility-group call when an SVC 202 is detected and initiates the abend. If access register 1 contains a valid ALET value, the compatibility-group macro may produce a code expansion that will attempt to modify the data/address space associated with the ALET value. This can result in a program check or an inadvertent space modification. If a program check does not occur during the macro code execution, CMS subsequently terminates the call with a X'1CD' abend code.

Effect of Data Space Support on Simulated Programming Interfaces

Certain macros provided by OS/MVS and DOS/VSE are simulated by CMS. This section discusses those macros.

OS/MVS and OS/VSAM Simulated Macro Interfaces

OS/MVS and OS/VSAM macros simulated by CMS are not AR-mode capable. The MVS set of macros provided are at an MVS/SP level, with the exception of the CALL, SYSSTATE, IHASDWA, and IHAEPIE macros, which are at the MVS/ESA level.

Calling OS/MVS simulated macros in AR mode results in abend code X'0F8' and reason code X'18'.

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.

DOS/VSE Simulated Macro Interfaces

z/VM abends DOS service calls made through DOS/VSE macros if they are issued in AR mode in an XC virtual machine. An abend code of X'1CD' indicates this.

Effect of Data Space Support on Existing Programs

Existing CMS application programs that use the CMS application programming interface are unaffected by data space support when they run within the scope of the specific interface. That is, applications that currently run on ESA or XA virtual machines can run successfully on an XC virtual machine. Further, a CMS program that is written to take advantage of ESA/390 architecture can run on an XC virtual machine with little or no change. Similarly, applications that are written to take advantage of z/Architecture can run on z/CMS in an XC virtual machine with little or no change.

For more information, see *z/VM: ESA/XC Principles of Operation* and *z/VM: z/Architecture Extended Configuration (z/XC) Principles of Operation*.

An application running in an XC virtual machine can exploit data spaces. To be able to operate directly on data within a data space, the application program must execute in AR mode on an XC virtual machine. New CMS application programs written to exploit data spaces can coexist with existing application programs in an XC virtual machine. New data space applications that have dependencies on existing, non-AR mode applications must transfer control to these applications in the following manner to ensure predictable results:

- Use CMS supervisor-assisted linkage (CMSCALL or AMODESW macros)
- Return to primary-space mode before using a branch transfer.

AR Mode Execution Considerations

Figure 34 on page 240 shows portions of a program, PGM1, executing in the USER1 XC virtual machine in AR mode. In this example, the DMSSPCC and DMSSPLA routines were called previously to create data spaces DataSpaceX and DataSpaceY and add them to the access list of the USER1 virtual machine.

Note: An AR mode program must set the access register associated with its base register to a 0 ALET value. This ensures that when an instruction uses the base register of the program to reference an address, the instruction references an address within the user's primary address space.

Once the data spaces have been created and addressability to them has been established, the program uses assembler instructions to manipulate data in them. The numbers to the left of the instructions in the figure correspond to the following steps.

1.
PGM1 issues the SAC 512 instruction to enter AR mode to manipulate data directly in the data spaces.

2.
It then issues the LAE instruction to establish general register 12 as a base register and to zero out the associated access register (AR12).

Next, it uses the LAM, SR, and USING assembler instructions to establish the base address for the data spaces.

3.
The LAM instruction loads the ALET of DataSpaceX into AR 2 and the ALET of DataSpaceY into AR 3. The ALETs were obtained using the DMSSPLA routine.

4.
The SR instructions are used to establish the origin address of the data spaces. Data spaces created in VM have an origin of 0.

5.
The USING statements identify the mappings of the data space areas.

The next set of instructions, the L, ST, and MVCs, show data movement among the primary address space and the data spaces.

6.
The L and ST place data from the primary address space into data space DataSpaceX.

7.
The first MVC then moves this same data from data space DataSpaceX to DataSpaceY.

8.
The second MVC moves the data from data space DataSpaceY to the USER1 primary address space at location DATOT.

Essentially, the character string of 'ABCD' originally located at DATIN in the primary space has been moved to DataSpaceX, to DataSpaceY, and finally, back to the primary space at location DATOT.

9.
The SAC 0 instruction takes the program out of AR mode and back to primary space mode.

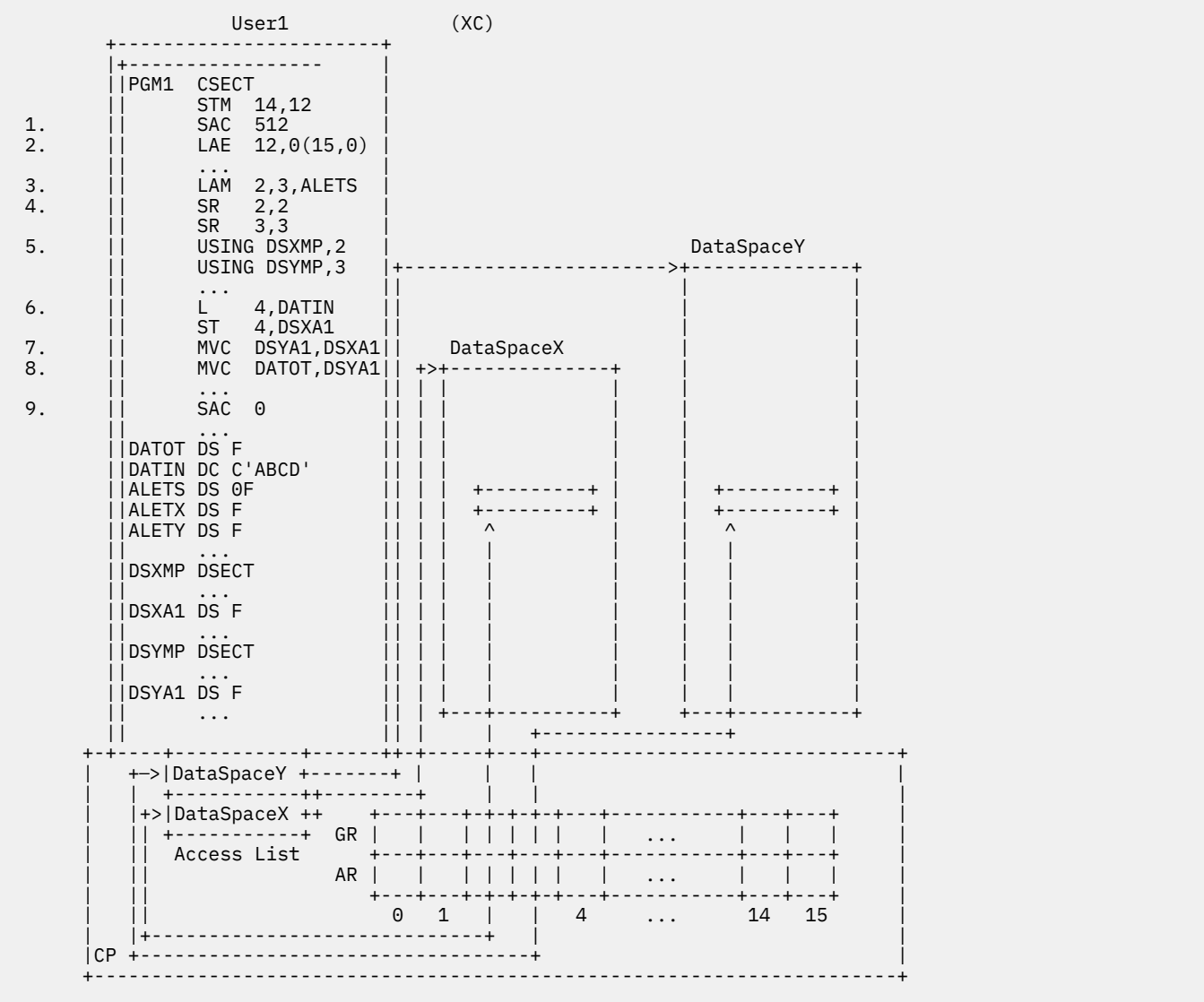


Figure 34. Access Registers and Data Space Addressability

Chapter 16. Your Applications and Data Integrity

This chapter describes the Coordinated Resource Recovery Services and how to design your applications to ensure data and system integrity within your application.

Introduction to Coordinated Resource Recovery Services

As people depend on computers to perform increasingly important and complex work for them, requirements for data and system integrity become more stringent. Coordinated Resource Recovery (CRR) provides a means of ensuring data integrity (consistency) for complex transactions and distributed applications.

Applications that do **not** write to more than one resource on a work unit need not be concerned with CRR. However, some applications need to update data in multiple places and be assured that all changes are made, or if that is not possible, that no changes are made. The application's data must always be in a consistent state.

CRR enables an application running on CMS to update multiple resources with integrity, whether they reside locally or remotely in an SNA network.

How CRR Works

To get this function, the application need only use ***protected resources*** (resources, such as the Shared File System (SFS), that participate in CRR). These protected resources utilize a repository controlled by a program called a ***resource manager*** to keep track of both the new and old versions of the data. Upon request, the resource manager of the repository can either commit the changes, making them permanent, or roll back[2] the changes, thereby restoring the original values.

Some applications need to communicate with other applications in other virtual machines or systems which themselves change data. In scenarios like this CRR also coordinates changes among multiple applications. This capability is provided through APPC/VM support for protected conversations (both in assembler and the Communications element of the Common Programming Interface (CPI) for high-level languages).

Recall from Chapter 12, “Manipulating SFS and Minidisk Files and Directories,” on page 129 that in CMS, work is divided into logical units of work. These logical units of work are sets of changes that can be viewed as units of recovery. The changes made by these logical units of work are associated by a work unit, which is identified by a work unit ID. After an application makes a set of changes (completes a logical unit of work) on a CMS work unit, it commits all changes associated with that work unit or rolls them all back. Changes done under other work units are unaffected. The point at which a commit or rollback is done is also known as a ***synchronization point*** (usually called ***sync point***). The set of changes made on a CMS work unit can also be referred to as a ***transaction***. Once the set of changes on a work unit has been committed, the application repeats the process with a new set of changes to data (another transaction). This type of application is commonly known as a transaction program (TP). Figure 35 on page 242 illustrates this concept.

[2] The term "rollback" (verb form "roll back") is generally used in CMS. The synonymous term in SAA and SNA is "backout" (verb form "back out"). Because of the relationship between CRR and SNA, the SNA term backout does appear in CMS documentation along with rollback.

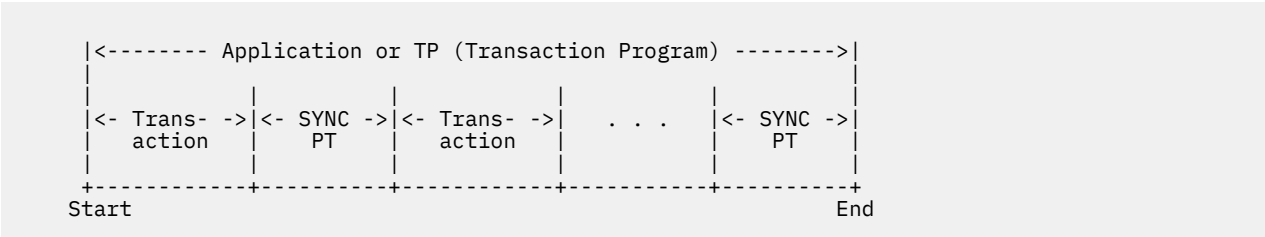


Figure 35. Relationship between an Application, Transaction, and Synchronization Point

An application is typically made up of transactions separated by sync points. A simple application looks like a single work unit to CMS. A complex application can have more than one transaction at the same instant (and therefore, more than one CMS work unit). For the purposes of this discussion, transaction and logical unit of work are synonyms.

Because transactions can change data, we want to be sure that the changes made can be recovered (made consistent with each other after a failure). This is accomplished by separating transactions with sync points where all recoverable changes are either made permanent or undone. A sync point looks like a single, atomic operation to the application. An application always ends with a sync point, which should be explicitly coded in the form of a commit (or rollback).

CMS provides the coordination process for the commit and rollback of data across multiple repositories through its synchronization point manager (SPM). CMS, through CRR, can coordinate up to about 200 resources (SFS file pools, other participating resources, and protected conversations) within a transaction (logical unit of work). The SPM accepts commit and rollback requests from the application and works with the affected resource managers, represented by resource adapters, to ensure a consistent state for the data.

Whenever an application accesses multiple resources and commits those resources, CRR creates a unique Logical Unit of Work Identifier (LUWID). The LUWID identifies the changes in all the various resources that are being coordinated. This is important to know because of the role that the LUWID plays in resource recovery following a failure.

CRR is implemented using a two-phase commit protocol. The first phase involves the SPM acting as coordinator by requesting all resource managers involved to get ready to commit the changes for the work unit. Each resource manager responds indicating whether the changes are complete. When the SPM receives a response from all the resource managers, it decides whether to commit or roll back the work unit. If all of the resource managers indicate that the changes can be made, the decision is to commit. If any reply indicates that a change cannot be made, the decision is to roll back. The decision is recorded in the CRR recovery server's log and that decision is then passed on to each resource manager to carry out. The recording of the decision in the CRR recovery server's log marks the transition from the first to the second phase of the commit. In the second phase, the SPM notifies all of the resource managers of the decision and the resource managers must then carry it out.

Note that, because of a severe failure, such as the SFS file pool server machine *going down*, the commit or rollback may not be completed at the time the application requests the commit or rollback. CMS provides an additional function, ***resynchronization***, that will complete the request at a later time, when contact is restored with all affected resources. This is automatic; your application does not have to be restarted in order for this to occur. Resynchronization may occur after the application ends.

Designing Your Application for Data Integrity

The section entitled "Data Recovery/Data Integrity" on page 24 contains some concerns about data integrity that you should address before starting to design your application. If you have not reviewed that section already, it would be well worth your while to look at it before continuing on here because it contains some important background that will help you decide what to include in your design.

Setting Up to Ensure Data Integrity

As we discussed earlier, applications are made up of transactions, or sets of changes to data. Transactions must be associated with work units. Your application can use the default work unit, but it is recommended for more complex applications that your application call the DMSGETWU (Get Work Unit ID) CSL routine to obtain a work unit. DMSGETWU gives you the control you need for complicated applications. DMSGETWU also allows you to specify a default transaction tag for the work unit.

A transaction tag contains a message up to 80 characters long that is written to the CRR recovery server's log to aid in the recovery from a failure. At sync point, this message is also available to be written to the logs of the resource managers involved in the work unit, which includes the logs of all SFS file pools that the application is connected to.

As mentioned earlier, your application can specify a transaction tag on the DMSGETWU routine. However, your application, whether it uses the default work unit or obtains a work unit using the DMSGETWU routine, can call the DMSSETAG (Set Transaction Tag) routine. The DMSSETAG (Set Transaction Tag) routine sets or changes a transaction tag at any time on the default work unit or another work unit.

Consider the following problem that occurs when the operator or administrator looks at the CRR or SFS log after some failure during sync point processing and asks: "What application was running?" If an operator or administrator is forced into a situation where they must recover data, how do they know what application was originally invoked?

The solution is to have your application tell CMS what to store on the log so that the CRR recovery server operator can determine what needs to be done during resynchronization. Each transaction (or logical unit of work) can have its own information (tag) saved on the log at sync point. Your application can specify a default transaction tag when it obtains a work unit and can set or change the transaction tag at any time. The transaction tag is written to the CRR recovery server log and then passed to the adapter exits (the interface to the resources being accessed) for use on their resource manager's log.

You can use transaction tags to communicate application-specific information to administrators performing CRR problem determination procedures. The exact content of the tag is left to the discretion of application programmers. It is up to you, the application programmer, to establish standards for its content. Transaction tags can also be extremely helpful for problem determination in a distributed environment. The ability to customize transaction tag information allows your application to return transaction-specific data.

Transaction tags are **not** automatically known between applications that communicate using protected conversations, however. For example, if an application, APPL1, is connected by a protected conversation to another application, APPL2, then the transaction tag that APPL1 sets is not logged in the CRR recovery server for APPL2. APPL2 must set its own transaction tag. If you want a transaction tag to appear in the CRR logs for all applications involved in a transaction, then you can pass the tag information as data through the protected conversations, thus allowing the receiving programs to create an identical transaction tag.

Examples of useful transaction tag information are:

- User ID of the end-user executing the distributed application
- Name of the application
- Type of activity being performed (add, change, delete, etc)
- Other key information.

A possible transaction tag might be:

```
BIGDEALS  SELLER1.VMCRR  Add  Sales-order=76235430
```

This transaction tag indicates that the user SELLER1 at node VMCRR was executing the BIGDEALS application, specifically adding sales order 76235430.

Another approach to transaction tags might be:

```
BIGDEALS  RECOVERY INFO=file_id_xyz
```

This transaction tag provides only the application name and a pointer to a file kept by the application which contains specific information, such as what steps to perform when recovering BIGDEALS and what resources were being updated. This is a more flexible way of using transaction tags, but it requires the operator to go to the file indicated to get further information.

Once the second phase of the commit is completed for a resource, the log data is typically discarded for that resource. It is possible in the event of operator intervention or a catastrophic failure on the part of one of the resources during resynchronization, that some of the log data needed to manually bring the data back to a consistent state may not be available. For example, say application A calls applications B and C, which update resources D and E, and F and G, respectively, as shown in Figure 36 on page 244:

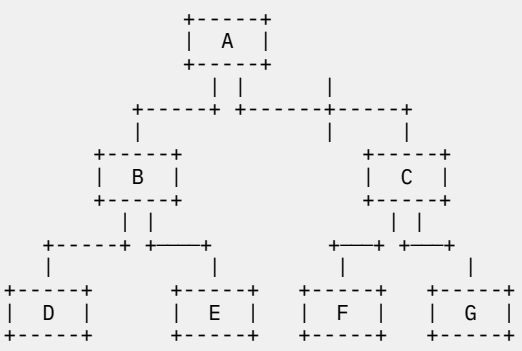


Figure 36. Hierarchy of Application Calls and Updates

During the second phase of the commit, even if resynchronization becomes necessary between A and C, the log records at B, D, and E are still erased when they complete the commit. Then, if the resynchronization process fails, and the resulting operator intervention results in a rollback for C, F, and G, the data is left in an inconsistent state (resulting in an error condition). Such a situation requires manual recovery, but the log data for B, D and E is no longer available, making the determination of the extent of data inconsistency very difficult at best. Therefore, if your application is performing critical transactions, it should keep its own record of each transaction until it is clear that the transaction has been successfully committed or rolled back.

Setting Synchronization Point Options

Another part of setting up is anticipating problems that could occur during sync point processing. The CSL routine DMSSSPTO (Set Synchronization Point Options) allows you to set certain options for the sync point. If you do not use DMSSSPTO to set any of the options, the system defaults will be used. The default settings are as follows:

WAIT
If a resynchronization is required, control is not returned to your application until resynchronization completes.

BACKOUT
If a protocol violation (notification of an inconsistency) is received from the sync point initiator while in prepared state (ready to commit), CMS will roll back the changes to the work unit(s) specified for the sync point.

SYNC
On certain requests the virtual machine will remain in a wait state until the request completes.

The choices you need to make depend on your application and the installation where it will run. The following discussion should help you understand what each choice of options means to your program.

WAIT/NOWAIT: When the default option WAIT is in effect, your application will enter a wait state until a return code is returned to your program after the resynchronization completes. This return code will indicate one of two possibilities:

- Recovery was successful as indicated by the return code (either a commit or roll back was completed), or

- A commit or rollback was performed, but the state may not be consistent due to a unilateral decision (possibly operator intervention) on the part of one or more resources to commit or roll back contrary to the global (CRR) decision.

In the first case, if a commit is indicated, your program can continue processing from that sync point. If a rollback is indicated, it will be necessary to try the transaction again. You will have to prepare for the second case (a possibly inconsistent state) in accordance with the type of application you are designing. Some situations may require that your program terminate, while other situations may allow for less drastic measures.

If your application must operate in an environment that dictates a different course of action than that just described (for example, delays of more than a few seconds are intolerable), you can set the NOWAIT option. This option means that resynchronization will still be tried, but if it is not successful on that first try, control is returned to your application with a resynchronization-in-progress reason code. This could be useful if your application processes transactions in an interactive environment. Your program can continue other processing (unless the system goes down), and the resynchronization will complete later, possibly even after your application ends. Alternatively, if the success or failure of the transaction in resynchronization could affect future transactions, your program could terminate.

In any case, you should consult your system administrator so you can be aware of computer center procedures for such things as recovery and operator intervention.

BACKOUT/COMMIT: If a protocol violation is received from the initiator (your protected conversation partner) of the synchronization point while in prepared state, CMS rolls back the changes to the specified work unit(s) by default. This is an attempt to ensure that recovery can be achieved quickly and reliably. It is generally safer to assume the worst in transaction processing. If this is not satisfactory for your application's environment, you can set the COMMIT option so that an attempt will be made to commit the changes in the event of a protocol violation on the part of the initiator of the sync point.

SYNC/ASYNC: The (default) SYNC option allows the SPM to communicate synchronously with a protected resource when it is more efficient to do so. Ordinarily, when the SPM needs to communicate with more than one protected resource, it does so asynchronously. There are times, however, such as when reading from one resource and writing to another, when the SPM is only communicating with one resource at a time, even though more than one resource is involved. If ASYNC is set, the SPM is forced to communicate asynchronously with that one resource. The SYNC setting allows the SPM to switch to synchronous communications when that is the better choice.

If your application performs multitasking, you should set the ASYNC option. This allows CMS to regain control immediately and to call DMSCWAIT (CRR Wait) for any requests to the CRR recovery server. Your application can then intercept the call to DMSCWAIT and dispatch another task (which will be using a different work unit ID). You must provide a replacement for DMSCWAIT to handle the waiting. See Chapter 17, “Writing a CRR Wait Routine for Multiuser Server Applications,” on page 251.

ALL/SINGLE: Should you decide to set any of the sync point options, you can also specify whether you want the settings to apply to all current and future work units or only to a specific work unit.

Committing (or Rolling Back) Changes

In order to make the changes permanent, the work unit making the changes must be committed. **IBM strongly recommends that your application issue an explicit commit as soon as the processing for a logical unit of work is completed.** You can also issue an explicit rollback when problems occur in the work unit; otherwise, an implicit commit is issued at end of command. With CRR, all changes to all protected resources made on the specified work unit(s) will be committed (or rolled back if all resources could not commit) when a commit routine of any of the protected resources is issued.

It is possible, however, that not every resource your program will access is capable of participating in CRR. This means that you must find out before coding your application, which resources participate in CRR and which ones do not.

If any of the resources are unable to participate in CRR, changes made to data controlled by those resources cannot be guaranteed to be consistent with the data controlled by protected resources (those participating in CRR). You must issue a commit for the nonparticipating resource(s) using the product-

specific commit routine. To reduce problems, design your application to issue resource-specific commits **first** to make permanent any changes to resources not participating in CRR. Dividing the changes for various resources into separate work units can make this task much simpler because your program can commit one work unit at a time. The preceding discussion applies when you want to roll back unwanted changes.

For all resources participating in CRR, your application can issue a commit or rollback using one of the following ways:

- Using the SRRCMIT (Commit) and SRRBACK (Backout) SAA resource recovery (also known as CPI resource recovery) routines
- Using the DMSCOMM (Commit) and DMSROLLB (Rollback) CSL routines
- Asking the participating resource to commit or roll back (SFS return codes are not available in this case).

The synchronization point manager takes care of propagating this signal to all protected resources.

An important consideration in deciding which routine to use for committing or rolling back a work unit is the amount of information available to your application in the event of an error or other problem. The DMSCOMM and DMSROLLB CSL routines return both a return code and a reason code to indicate what problem occurred. The SRRCMIT and SRRBACK routines provide a return code only, while other participating resources may provide a return code, error information, or both. See the documentation on the individual participating resource for details on possible return codes.

When all of the work in a work unit has been completed, your application can return the work unit to CMS using the DMSRETWU (Return Work Unit ID) CSL routine. Returning the work unit just indicates that the system may free its storage associated with that work unit. If the work has not been committed, a coordinated commit is issued automatically. SFS files and directories that are open under the work unit are closed and protected conversations associated with the work unit are deallocated. If you use this routine, you should use the DMSPOPWU (Pop Default Work Unit ID) CSL routine to remove the work units from the stack prior to returning them, because the work unit stack is not checked or modified by DMSRETWU.

So far, we have only discussed the explicit ways of committing changes, although implicit commits are also possible. Whenever CMS does an implicit commit or rollback, it does a coordinated commit or rollback. An implicit commit is done at end of command, for example, at the `Ready;` message. If the commit does not succeed, however, an implicit rollback will be performed, and your application may not be able to find out about it. IBM strongly recommends that your application always issue an explicit commit as soon as the processing for a logical unit of work is complete. Remember that changes made to a file are not actually reflected in the file until they are committed. For more details with regard to committing changes in an application program, see Chapter 12, “Manipulating SFS and Minidisk Files and Directories,” on page 129.

A Few Notes on Rollbacks

A rollback can occur at any time, but synchronization point processing is when it is most likely because that is when changes are committed and when problems making the changes permanent are encountered. Note, however, that rollbacks can also occur at times other than during sync point processing. For example, a failure during a write attempt can cause a rollback before sync point processing. As a result of such a failure, your application will receive a return code indicating one of two situations:

- A coordinated rollback must be performed on the current work unit. A resource, such as a protected conversation, cannot issue rollbacks (or commits). The resource relies on the application to issue a rollback when one is required.
- A coordinated rollback has already been performed. A resource manager, such as SFS, automatically issues a rollback.

Other participating resources may have different rules. As an application writer, you must reference the protected resource documentation for the correct application action.

If an application issues a commit but gets a return code indicating a rollback on that commit, this means that all work was rolled back. The application is not required to issue a rollback in this case.

Tracking Down Errors

How does an application find out why and where a commit was rolled back, why a rollback had problems, or if warnings occurred? Although your application will receive a code from the commit, this may not be enough to determine what actually went wrong and what course of action (if any is possible) should be taken to correct the problem. It is possible to get detailed information about errors by using the DMSGETSP (Get Synchronization Point Errors) CSL routine to retrieve error blocks saved by the synchronization point manager. You can use this error information to identify which resource encountered a problem or is involved in resynchronization processing.

Error blocks are saved for all warnings and errors detected since the last commit or rollback for a work unit. The format and content of the error blocks is resource dependent and is, therefore, not specified in the CRR architecture. However, error blocks should contain the resource ID for which the error occurred and perhaps an error code identifying the cause of the problem (for example, SFS supplies reason codes).

Your application can identify a product by a component ID, adapter exit name, or both. For IBM products, the component ID can be found in the *Programming Systems General Information Manual*. You need to identify the product in order to determine the format of the error block, which is specified in the documentation of each product.

z/VM provides two CSL routines to help you interpret error blocks—DMSWUERR (SFS Wuerror Deblocker) and DMSPCAER (Protected Conversation Adapter Errors). DMSWUERR extracts the error information from error blocks created by SFS. DMSPCAER extracts the error information from error blocks created when protected conversations encounter problems during sync point. See the *z/VM: CMS Callable Services Reference* for details on the DMSWUERR and DMSPCAER routines.

Generally, if you are using more than one resource type, your program should:

1. Call DMSGETSP to determine the error block length
2. Use the length obtained from the call to prepare to retrieve the error data
3. Call DMSGETSP (possibly in a loop) to retrieve the error blocks
4. Handle the errors.

Figure 37 on page 248 shows an example REXX exec for iteratively retrieving sync point error blocks. This example assumes all of the error blocks are the same length, but this may not always be a safe assumption.

The SPM keeps track of the length of each error block and resets this length value for all error blocks for a work unit at the start of each synchronization point. This length is the value that is returned in the *actual_error_data_length* parameter of the DMSGETSP (Get Synchronization Point Errors) CSL routine.

```
/*********************************************************************/
/*   This REXX exec provides an example of a typical call to DMSGETSP. */
/*********************************************************************/
/*                        Get Sync Point Errors                      */
/*********************************************************************/
retc = 0
reason = 0
data = copies(' ',284)            /* Initialize Data Area to size needed for SFS errors. */
data_length = length(data)             /* Get the length of the data    */
eb_length = 0                          /* Initialize usable data length */
cursor = 0                             /* Initialize the cursor to zero */
workunit = 0                      /* Get error information for the default work unit.    */
exitname = copies(' ',8)               /* Initialize exit name          */
exitname_length = length(exitname)     /* Get its length                */
resid = copies(' ',9)                  /* Initialize resource component */

/* Call for the 1st error block                                       */
call csl ('DMSGETSP       retc            reason ',
          'data           data_length     eb_length ',
          'cursor         workunit        exitname ',
          'exitname_length       resid')

If (retc = 4) & (reason = 90278)       /* Check the call results     /
  Then Do
    Say 'There are no syncpoint errors to retrieve'
    End

  Else Do
    /* There are either some syncpoint errors, or there was an unexpected error on    */
    /* the call to DMSGETSP.  Loop only if there are syncpoint errors to retrieve.    */
    Do While ((retc = 0 | retc = 4) & (reason ¬= 44040))

      If (retc = 4) & (reason = 90271)
        Then Do
          /* Buffer used to hold the error block is not big enough, create a buffer   */
          /* the size of the error block length set by the previous call to DMSGETSP. */
          retc = 0
          reason = 0
          data = copies(' ',eb_length)
          data_length = length(data)
          /* Reinitialize the cursor.  This will reset the position in the error     */
          /* block list, so some errors might be retrieved for a second time.         */
          cursor = 0                   /* Reinitialize the cursor      */
          End
        Else Do
          /* An error block has been successfully returned by DMSGETSP.               */
          /* The exit name and resource ID identifies the format of error block.      */
          /* Interpret the contents of the error block.                               */
          End

      /* Get the next error block                                     */
      call csl ('DMSGETSP       retc            reason ',
                'data           data_length     eb_length ',
                'cursor         workunit        exitname ',
                'exitname_length       resid')
      End                                /* End of Do While            */

    If reason = 44040
       Then
          Say 'All syncpoint errors have been successfully returned'
       Else
          Say 'DMSGETSP executed with return code' retc,
              '  and reason code' reason
    End
```

Figure 37. Example of a REXX Exec Used to Iteratively Retrieve Error Blocks

Notes for Distributed Application Programs

An application may be distributed, meaning it has multiple parts that communicate through a conversation. To take advantage of CRR, use protected conversations wherever possible. Whenever a sync point is reached in any part of a distributed application using protected conversations, the system requests the other parts to synchronize by issuing a commit or rollback. In other words, if your application is communicating with other applications in an SNA network, any time it receives a notification through a protected conversation to commit, it must issue a commit routine (or roll back if it cannot commit). If the request is to roll back, it should issue a rollback routine.

To understand how this works, recall that in CMS, a work unit is used to associate a set of changes that must be committed (or rolled back) in unison (logical unit of work). The situation gets more interesting when the application is distributed, because each part of the application will have its own work units and some of them may need to be committed in unison with those of other parts of the application. The protected conversation plays a vital role in the coordination of the work in these work units by providing the link between them. The LUWID identifies the protected conversation and all protected resources

associated with it. This LUWID is comprised of three components: the fully qualified logical unit (LU) network name; the instance number, which is unique at the LU that creates it; and the sequence number, which is incremented by 1 following every sync point.

When a protected conversation is established, an LUWID and CMS work unit ID are associated with it. Then, when the conversation is accepted, its LUWID is associated with a new work unit in the target application. Figure 38 on page 249 illustrates how CMS work units and protected conversation LUWIDs are related.

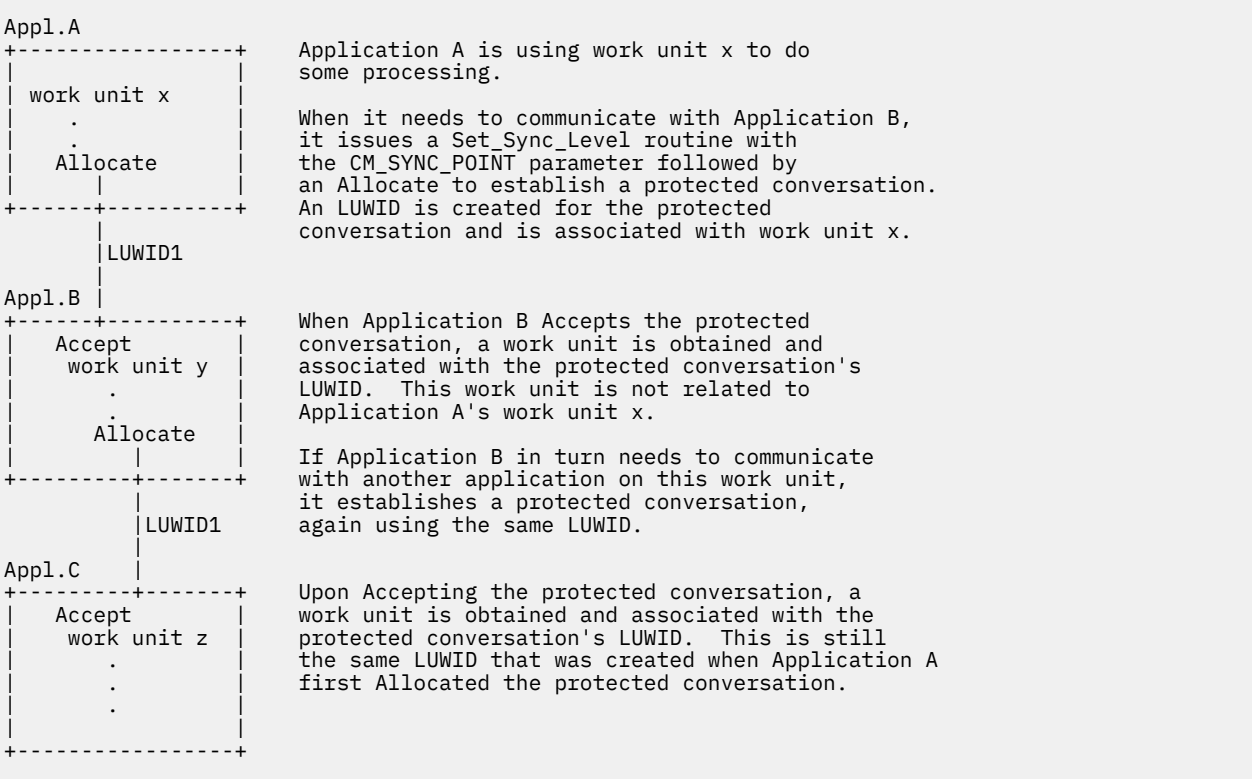


Figure 38. Relationship between CMS Work Units and Protected Conversation's LUWID

The LUWID ties the CMS work units together, so that whenever a sync point is reached and a commit is issued on a work unit, the partners are notified and are requested to issue a commit of the work unit with which they have associated the protected conversation. At this point, the system takes the responsibility to coordinate the commit of all data for all parts of the application such that all parts either are committed or rolled back. Should a rollback be required, even due to a failure outside your application, protected conversations can put your application in a *backout required* state. This means that your application should issue a rollback on the work unit associated with the conversation.

It is also possible for one of the partners to have protected conversations that it initiated with other programs under the same work unit, so that these other programs would participate in the sync point. A program may initiate multiple conversations on a work unit, but may accept only one.

When using protected conversations, your application may get errors that pertain only to those conversations. You can use the DMSGETSP CSL routine to retrieve the detailed error passback blocks that contain extended error information (as described in “Tracking Down Errors” on page 247) and then use DMSPCAER CSL routine to parse the blocks of data for a particular conversation. This can be particularly useful when debugging the communications sections of your application. DMSPCAER and DMSGETSP are described in the *z/VM: CMS Callable Services Reference*. You can also use the DMSGETER (Get My Errors) CSL routine to retrieve the detailed error passback blocks for use with DMSPCAER. This routine is described in the *z/VM: CMS Callable Services Reference*. DMSGETER allows you to specify the resource component ID and the exit name of the resource adapter for which you want extended error data. This routine returns any detailed error blocks saved by the specified resource adapter since the last sync point. DMSPCAER then uses the detailed error blocks to extract and return the extended error information for the specified conversation.

If your application does work unit management with multiple work units, as a server might, each work unit can be considered as a separate instance of the application. By manipulating work units, your application is doing some things that SAA designates as being the responsibility of the logical units. If you want to make your application portable by using only SAA interfaces, you must refrain from doing any work unit management. In the example outlined in Figure 38 on page 249, CMS does the work unit management by getting a work unit during Accept. When the conversation initiator deallocates the conversation, the partner application must end, causing CMS to return the work unit. A program that does not need to be portable could issue DMSRETWU (Return Workunitid) CSL routine for the conversation work unit. DMSRETWU is not an SAA interface.

For information about setting up protected conversations, see *Common Programming Interface Communications Reference (https://publibfp.dhe.ibm.com/epubs/pdf/c2643999.pdf)* and the *z/VM: CP Programming Services*. For a sample scenario using CRR to synchronize multiple updates, see "Scenario 3: Synchronizing Multiple Updates" on page 511.

Chapter 17. Writing a CRR Wait Routine for Multiuser Server Applications

CRR tolerates but does not exploit CMS multitasking capabilities, so the routine it uses to wait on asynchronous requests, the CMS-supplied CSL routine DMSCWAIT, causes CMS to go into a wait state. This means that no work can be done on the virtual machine until the request completes. Therefore, if you are writing an application that will perform multitasking, like a multitasking server for a shared resource, you will want to provide your own routine for waiting on an event. This will avoid making all clients of the shared resource wait on the event requested by one client.

Asynchronous Processing in CRR

Before discussing how to write a wait routine, however, let's look at how the synchronization point manager (SPM) handles asynchronous processing, so we can understand how to take advantage of CRR's tolerance for multitasking.

When the SPM receives an asynchronous request, it starts the asynchronous process and then calls DMSCWAIT to wait for the request to complete. While the SPM—and consequently the virtual machine on which it is running—waits, the request is processed. Once the request has been completed, the interrupt handler for the module that processed it calls the DMSMARK CSL routine to allow the SPM to resume processing. (DMSMARK is described in the *z/VM: CMS Callable Services Reference*.) The SPM then checks the results of the asynchronous request and continues its processing. Figure 39 on page 251 outlines this sequence.

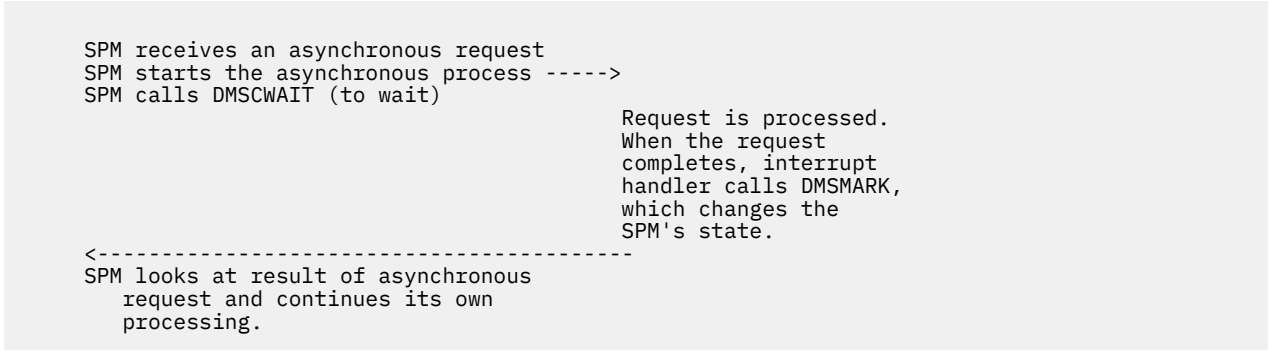


Figure 39. Asynchronous Processing Sequence in CRR

Requests to the SPM are identified by a special integer value, called a request ID, that is associated with the work unit originating the request. The SPM calls DMSCWAIT with this request ID.

Tolerance for multitasking during asynchronous communications can be achieved by exploiting the following CRR operating characteristics:

1. Whenever CRR must wait on an asynchronous event, it uses a replaceable system function, the DMSCWAIT CSL routine, to wait for the request to complete. For example, a resource adapter may be processing a coordination exit asynchronously.
2. By specifying the ASYNC option on the DMSSSPTO (Set Synchronization Point Options) CSL routine, you can indicate that CMS is to perform SPM processing asynchronously and call DMSCWAIT whenever it waits.

Multitasking Scenario

There are two categories of multitasking server applications, those that use CMS threads and those that implement their own subdispatching scheme. For these cases, the approach to using a DMSCWAIT replacement is similar. The CMS multithreaded application can use semaphores to wait, thus allowing

other threads to be dispatched. It would associate a semaphore with each work unit and wait on the corresponding semaphore. A thread that finds that the request is complete would signal the semaphore to awaken the thread, which would return to the caller of DMSCWAIT. The subdispatching application would suspend the task that was called for the DMSCWAIT and dispatch another.

Below is a simple scenario that applies to both and uses the generic term *task* to refer to either a CMS thread or an application managed dispatchable unit. Your multitasking application gets a work unit for a task and dispatches it. When the task reaches a sync point (for example, it issues a commit), the SPM gets control so it can provide coordination if necessary. Because you previously issued DMSSSPTO with ASYNC specified, the SPM calls DMSCWAIT when waiting for asynchronous events. Your replacement for DMSCWAIT is actually called, so it can suspend the task making the request and associate the request ID passed on DMSCWAIT with that task. Before your program dispatches another task, however, it should issue a DMSCHECK NOWAIT for the request ID passed on DMSCWAIT. If the request has completed, the task just suspended can be resumed. Otherwise, another task can be dispatched. If no tasks are ready, your program should issue a DMSCHECK WAIT, specifying 0 for the *requestid* parameter.

Figure 40 on page 252 illustrates the flow of control between a multitasking application and the SPM.

```
Multitasking Application
    Work unit for each task
    Dispatch Task1
       do some processing
       issue a commit ------------------------> SPM

                                                  issue DMSCWAIT
    <----------------------------------------
    Suspend Task1
    Dispatch Task2
       do some processing
       issue a commit ------------------------> SPM

                                                  issue DMSCWAIT
    <----------------------------------------
    Suspend Task2
    Dispatch Task3
⋮
    Dispatch Task1 (after its request ID is returned on DMSCHECK)
⋮
```

Figure 40. Flow of Control between a Multitasking Application and the SPM

The following steps outline the operation of a replacement for the DMSCWAIT CSL routine for a subdispatching application. See Figure 41 on page 253.

1. Call the application's context switching routine passing the request ID received as an input parameter on the call to your replacement for DMSCWAIT.
2. If Suspend_Task was OK, then return OK.
3. Otherwise, return an "invalid request ID" code.

Note:

1. The application **must** maintain a one-to-one-to-one relationship between tasks and CMS work units and request IDs. Therefore, the application (dispatcher) will have to get a work unit for each task that it dispatches.
2. The application should map one of its tasks to the request ID passed as input on DMSCWAIT. Note that more than one task may become ready before the request completes.

3. For a subdispatching application, its Dispatcher must eventually:
   a. Issue a check (DMSCHECK) with NOWAIT specifying this SPM request ID for the *requestid* parameter, or
   b. Receive this SPM request ID as output from DMSCHECK with *requestid* specified as 0 (any).

   When ready to run, resume this task and return an OK return code.

   For an application using multiple CMS threads, at processing breakpoints it should also issue the check (DMSCHECK) as above and signal the semaphore corresponding to the request ID. This resumes the thread on which DMSCWAIT was entered, which returns an OK return code.

```
Context_Switcher

    Suspend_Task

    1. Save caller's registers and return address

    2. Associate request ID with suspended task

    3. Call Dispatcher -------->

                                              Dispatcher

                                                1.  Issue DMSCHECK with NOWAIT and
                                                    the request ID from DMSCWAIT specified
                                                    to see if the request has completed.

                                                2. CASE of
                                                    1)  Request completed
                                                        Continue task just suspended
                                                    2)  Request not completed, but ready task
                                                        Dispatch it
                                                    3)  No ready tasks
                                                        Call DMSCHECK with WAIT option
                                                        and request ID of 0
                                                        Dispatch ready task
                                                    end case.
                                                3. Return
    4. Return  <-----------------
```

Figure 41. Context Switching Routine for Replacement of DMSCWAIT

Replacing DMSCWAIT

You can replace the IBM-supplied CRR Wait function by writing your own CSL routine, naming it DMSCWAIT, and loading it to override the IBM-supplied version. For general information about writing a CSL routine, see the *z/VM: CMS Application Development Guide for Assembler*. For information about the CSLENTRY, CSLGETP, and CSLEXIT macros used in writing a CSL routine, see the *z/VM: CMS Macros and Functions Reference*.

Exit Routine Parameters

The IBM-supplied CSL template file for the DMSCWAIT routine is DMS2OW TEMPLATE, shown in Figure 42 on page 253. This file identifies the routine's input and output parameters. The general format of a CSL template file is described in the *z/VM: CMS Macros and Functions Reference*.

```
3 3                     3 parms maximum, 3 required
SBIN  4  OUTPUT         Return Code
SBIN  4  OUTPUT         Reason Code
SBIN  4  INPUT          Requestid
```

Figure 42. DMS2OW TEMPLATE File

DMS2OW TEMPLATE contains templates for three required parameters:

1. The ***return code*** from DMSCWAIT. Your routine must return one of the following values:

Value
Meaning

0
The operation was successful.

8
The operation was unsuccessful.

2. The ***reason code*** from DMSCWAIT. The following value is defined:

   Value
   Meaning

   90216
   Invalid request ID. There is no such request ID.

3. The ***request ID*** that specifies the asynchronous events request to be waited for.

Making Your Exit Routine Available

You must arrange to have your version of DMSCWAIT called instead of the IBM-supplied module in VMLIB. You can do this by putting your DMSCWAIT in your own callable services library:

1. Generate a TEXT file for your routine.
2. Create a CSL control file for your library. Although you can use any file ID for the control file, you might want to adopt the convention of using the name of your library and the file type CSLCNTRL. For example, if you plan to name your library MYLIB, name the control file MYLIB CSLCNTRL.

   In the control file, create a ROUTINE record that identifies the CSL routine name, the TEXT file name, and the CSL template file name. For example, your ROUTINE record might look like this:

   ```
   ROUTINE DMSCWAIT DMSCWAIT DMS2OW
   ```

   The complete syntax of a ROUTINE record is described in the *z/VM: CMS Application Development Guide for Assembler*.
3. Use the CSLGEN command to build your callable service library. You can store your library on a minidisk or in a saved segment. For example, to build MYLIB on a minidisk accessed as file mode A, enter:

   ```
   cslgen dasd mylib from mylib
   ```

   For more information about the CSLGEN command, see the *z/VM: CMS Commands and Utilities Reference*.
4. Update the multitasking server's PROFILE EXEC:
   a. Add a command to access the minidisk on which your library resides. (This is not necessary if your library is in a saved segment.)
   b. Add an RTNLOAD command to load your routine. For example, to load DMSCWAIT from the MYLIB library, add:

      ```
      rtnload dmscwait (from mylib system
      ```

      You do not need to add a GLOBAL command because the library is specified in the RTNLOAD command. You do not need to add an RTNDROP command either; your own DMSCWAIT overrides the IBM-supplied version once the RTNLOAD command is executed. For more information about the RTNLOAD command, see the *z/VM: CMS Commands and Utilities Reference*.
   c. If applicable, release the minidisk on which your library resides.

Chapter 18. Getting a Resource Manager to Participate in CRR

This chapter is intended to help programmers with product development responsibility who are writing code to enable a resource manager to participate in CMS Coordinated Resource Recovery (CRR).

What Is CRR Participation?

The part of a resource manager that resides in the application's (user's) virtual machine and provides the communications link between the application and the separate resource manager machine is called the **resource adapter**. To participate in CRR, you must write additional code for your resource adapter and your resource manager. This chapter describes the additional function you must provide.

A single resource adapter might be able to handle multiple resources. For example, CMS provides a resource adapter that supports multiple SFS file pool servers.

Note: Typically, the link between the resource adapter and its resource manager is established through *nonprotected* APPC/VM conversations or some other mechanism that does not support coordination. However, a variation of CRR participation is possible in which the resource manager and resource adapter use *protected* APPC/VM conversations, which do support coordination. This variation has some unique rules. For more information, see “Using Protected Conversations” on page 301.

Participation in CRR is the interaction of the resource manager and resource adapter with two parts of CRR:

- The resource adapter interacts with the CRR synchronization point manager (SPM).

  The SPM is the part of CRR that is loaded into the application's virtual machine (with CMS). The resource adapter must register the resource with the SPM. During synchronization point (sync point) processing, the SPM drives exits to registered resource adapters for the various sync point functions. You must write one or two CSL routines to handle these exit calls. Your resource adapter can also use these exits to do additional processing.
- The resource manager interacts with the CRR recovery server.

  The CRR recovery server is a virtual machine that provides CRR logging and recovery functions. The resource manager must establish communications with the CRR recovery server to exchange log names and other data needed if resynchronization processing becomes necessary.

  Note: A resource manager that supports a simple (one-phase) commit and registers with CRR as the only write-mode resource permitted on the work unit can participate in CRR *without* a CRR recovery server. This type of participation is called "limp mode". Operating in this mode makes the resource more available to the application, but could cause some system performance degradation. In this mode, only one protected resource on a CMS work unit can be updated, and protected conversations are not allowed. You can still access SFS file pools in this mode, but you can commit files with integrity only within the *same* file pool.

For examples of the communication flows, see Appendix I, “CRR Communications Examples,” on page 583.

CRR Participation Requirements

For an application to call CRR services, the application must:

- Use resources that have the support needed to participate in CRR
- Explicitly or implicitly call a CRR commit or rollback.

To participate in CRR, a resource must have:

- A resource manager that supports:
  - The CRR two-phase commit process and CRR resynchronization
  - APPC conversations (SYNC_LEVEL=CONFIRM) for CRR recovery server communications
  - Exchange Log Names and Compare States APPC general data stream (GDS) variables (modified).
- A resource adapter that:
  - Registers the resource with the SPM
  - Understands the exit interface with the SPM, which consists of exit calls for the following sync point functions:
    - Precoordination
    - Coordination (two-phase commit)
    - Postcoordination
    - End of work unit
    - Backout required.
  - Translates resource manager-specific commit or backout requests (if Brand X has its own verbs) into CRR commits or rollbacks
  - Reflects to CRR any resource errors detected outside of a sync point.

Logging

Logging of data is a central requirement for participation in CRR. The SPM, the CRR recovery server, and the participating resource manager all have logging responsibilities. Logging by the SPM and the CRR recovery server is described in the *z/VM: CMS File Pool Planning, Administration, and Operation*.

There are two types of information that the resource manager must log:

- Resource changes being coordinated through CRR, in case the changes need to be recovered
- Information about the transaction, such as the transaction tag and the logical unit of work identifier (LUWID). The SPM passes these values to the resource adapter in the exit call.

The resource manager may also decide to put into the log other information that it must save, such as the CRR recovery server's LU name (locally known LU name and fully qualified LU name), transaction program name (TPN) or resource ID, and log name.

The resource manager can give its log any name, as long as the name uniquely identifies the current version and is not longer than 64 bytes. For example, the resource manager could use the cold start (log initialization) time stamp as the name of the log. The resource manager also might want to provide one or more facilities to reconfigure the log. For example, SFS provides the FILESERV LOG command.

Resource Adapter Interface with the SPM

The resource adapter's interface with the SPM consists of:

- A group of IBM-supplied CSL routines that the resource adapter calls for SPM functions. These routines, listed in Table 30 on page 256, reside in the VMLIB callable services library.

Table 30. CSL Routines the Resource Adapter Calls for SPM Functions

Routine	Description
DMSCHREG	Changes resource registration values.
DMSGETER	Retrieves error blocks containing warning and error data.
DMSGETRS	Gets information about the CRR recovery server.
DMSMARK	Marks the completion of an asychronous event.

Table 30. CSL Routines the Resource Adapter Calls for SPM Functions (continued)

Routine	Description
DMSREG	Registers a resource and its adapter with the SPM.
DMSSETR	Tells the SPM that a resource has had a backout (rollback) or failure.
DMSUNREG	Deletes the registration of a resource.

For more information about the format and content of these CSL routines, see the *z/VM: CMS Callable Services Reference*.

- One or two CSL routines that you write as part of your resource adapter to provide sync point exits for the SPM. After your resource adapter registers for CRR participation, the SPM drives these exits to the resource adapter for the sync point functions shown in Table 31 on page 257. The exits provide opportunities for functional customization by registered resource adapters.

  Table 31. Resource Adapter Exits

Function	Description
Precoordination	Called to make sure that the resource is ready for a sync point.
Coordination	Called to include the resource in a sync point.
Postcoordination	Called to clean up after a sync point.
End of work unit	Called for cleanup processing before the work unit ends.
Backout required	Called to put the resource in a state such that a backout is required.

  All exits to the resource adapter can be driven through a single CSL routine. However, because the backout-required exit is driven from the interrupt handler, which restricts the processing allowed in that exit, you can write a separate CSL routine to handle the backout-required function. Information about writing the exit routines is provided later in this chapter.
- The ADAPTRC macro, supplied by IBM in the DMSGPI MACLIB, which defines the constants (sync point functions, sync point actions, return codes, and response codes) used in the resource adapter exits. The names of the constants begin with the letters ADA. For more information about the ADAPTRC macro, see the *z/VM: CMS Macros and Functions Reference*.

Registering a Resource for CRR

To participate in CRR, a resource must be registered with the SPM. The resource adapter registers the resource for a particular CMS work unit by calling the DMSREG (Resource Adapter Registration) CSL routine. Data items that the resource adapter supplies in DMSREG include:

- The adapter token that the resource adapter uses to identify the resource (because the resource adapter might be handling multiple resources)
- The names of the resource adapter's exit routines

  Note: The exit routines must be loaded into the application virtual machine (using the RTNLOAD command) before you can use DMSREG to register.
- The resource manager's fully qualified LU name and TPN
- The resource recovery TPN or PIP data
- The CMS work unit ID
- Various flags to indicate the type of registration (see “Setting the Registration Flags” on page 259).

This is not intended to be a complete list of DMSREG parameters. Some of the DMSREG parameters are optional; many have default values. For complete information about the format and content of the DMSREG routine, see the *z/VM: CMS Callable Services Reference*.

The output of each registration is a unique registry token that the SPM returns to the resource adapter. The resource adapter must save the registry token and use it in other CRR routines to identify the specific instance of registration (resource and resource adapter pair for the CMS work unit). A resource adapter handling multiple resources or multiple CMS work units may register many times.

Getting Information about the Resource Manager

Before calling DMSREG, the resource adapter might first have to get the following information about the communications link with the resource manager:

- Local (resource adapter's) fully qualified LU name
- Remote (resource manager's) fully qualified LU name
- Mode name (conversation characteristics)
- Resource manager's TPN
- Session instance ID (optional DMSREG parameter)
- Access user ID (optional DMSREG parameter).

How the resource adapter gets these values depends on whether the resource adapter is using CPI Communications (also known as the SAA communications interface) or the APPC/VM assembler programming interface to communicate with its resource manager.

Note: If the resource adapter and resource manager are on different processors, you should set up a CMS communications directory (COMDIR) file before using either method.

Using CPI Communications (SAA Communications Interface)

You can get the values you need (including the lengths of the variables) by calling the CSL routines indicated in the following list.

Value	CPI Communications (SAA Communications) Routine
Local fully qualified LU name	XCELFQ (Extract Local Fully Qualified LU Name)
Remote fully qualified LU name	XCERFQ (Extract Remote Fully Qualified LU Name)
Mode name	CMEMN (Extract Mode Name)
TPN	XCETPN (Extract TP Name)
Access user ID	XCECSU (Extract Conversation Security User ID)

For more information about these routines, see the *Common Programming Interface Communications Reference (https://publibfp.dhe.ibm.com/epubs/pdf/c2643999.pdf)*.

Using the APPC/VM Assembler Programming Interface

You can get the values you need (including the lengths of the variables) through the following sequence of macros.

1. APPCVM CONNECT establishes a communications path.

   Note: You must set up an APPC/VM parameter list before using this macro.
2. CMSIUCV CONNECT requests CMS to perform a CONNECT. The connection complete extended data returned by this macro includes:
   - Local fully qualified LU name
   - Remote fully qualified LU name
   - Access user ID

- Session instance ID.

3. CMSIUCV RESOLVE places the results of a COMDIR symbolic destination name resolution into the APPC/VM parameter list and the connection parameter list extension. These results include:
   - Mode name
   - TPN.

For information about the APPCVM macro, see *z/VM: CP Programming Services*. For information about the CMSIUCV macro, see the *z/VM: CMS Macros and Functions Reference*.

Getting Information about the CRR Recovery Server

Before the first sync point (specifically, before the end of the first precoordination exit call from the SPM) the resource manager and the CRR recovery server must exchange log names and other information to make sure they have consistent log information about the transactions (LUWIDs) that might require resynchronization processing. This is called resynchronization initialization. To establish communications with the CRR recovery server, the resource manager needs certain information about the CRR recovery server, such as its locally known LU name and TPN.

When the resource adapter allocates the conversation with the resource manager, the resource manager receives connection pending extended data (also called allocate data). This data includes the mode name and the "connect back" locally known LU name. The resource manager must save this information to use when allocating the conversation with the CRR recovery server. (The CRR recovery server has the same locally known LU name as the resource adapter because they are always on the same processor.) For a description of this flow, see Appendix I, “CRR Communications Examples,” on page 583.

To get the CRR recovery server's TPN, the resource adapter can call the DMSGETRS (Get Recovery Server Information) CSL routine and then pass the value to the resource manager. Although this task is not *required* before registration, as part of the registration process you may want to have the resource adapter call DMSGETRS to get the information, have the resource adapter pass the information to the resource manager, and then have the resource manager do the exchange of log names. If the resource manager is on the same system as the resource adapter, the resource manager can call DMSGETRS.

Note: If the resource adapter calls DMSGETRS and receives a message that the CRR recovery server is unavailable, registration is still possible if the resource supports simple commits and is the only write-mode resource on the work unit. This type of CRR participation is called "limp mode". No exchange of log names is necessary (or possible). Protected conversations are not allowed. Although the resource is more available to the application in this mode, there could also be a system performance degradation.

For complete information about the format and content of the DMSGETRS routine, see the *z/VM: CMS Callable Services Reference*. For more information about exchanging log names, see “Exchanging Log Names” on page 287.

Setting the Registration Flags

To indicate the type of registration the resource adapter is requesting, you can set the following optional flags in the DMSREG routine. (These parameters all have default values.)

- The ***simple-commit flag*** indicates whether the resource adapter and resource manager can support a simple (one-phase) commit call. Whenever possible, you should set this flag ON (the default) to reduce processing.

  If only one resource adapter is registered for write mode on the work unit (see the description of the write-mode flag), and no resource adapter is registered with the simple-commit flag set OFF, the SPM can optimize for a simple commit rather than a two-phase commit. In that case, because no resynchronization activity is needed, no logging is done by the CRR recovery server or the resource manager.

  If the simple-commit flag is set OFF, CRR does a two-phase commit regardless of the setting of the write-mode flag.

- The ***write-mode flag*** indicates whether the resource should be treated as if there are updates to be committed or backed out.

  If you set the flag OFF (the default), the SPM and the CRR recovery server do not do any logging for this resource. The resource is still called during coordination processing. However, if a problem occurs during coordination, no resynchronization processing is done.
- The ***single-writer flag*** indicates, when the write-mode flag is ON, whether this is the only write-mode resource permitted for the work unit. The default setting is OFF.

  For resource managers that do not support the two-phase commit protocol (such as a VM/SP Release 6 SFS file pool server), you should set this flag ON and also set the simple-commit flag ON.
- The ***function flags*** indicate the sync point functions (precoordination, coordination, postcoordination, end of work unit, and backout required) for which you want the SPM to call an exit to the resource adapter. The default is to exit for all of the functions.

  Under most circumstances, you should leave the end-of-work-unit flag set ON so the SPM exits to the resource adapter to clean up and unregister when the work unit completes. Your cleanup processing could include such things as releasing storage and severing paths to your resource manager. You must unregister as part of the end-of-work-unit processing.

  Note: If your resource manager is participating in CRR as an "interested party" that needs to know only when sync points occur and what their outcome is, you can register in "listening mode" by setting the postcoordination flag ON and setting all the other function flags OFF.

Changing Registration Values

The DMSCHREG (CRR Change Registration) CSL routine allows the resource adapter to change the following registry values without having to unregister the resource and reregister:

- Write-mode flag
- Single-writer flag
- Function flags
- Recovery token.

For example, if the resource is not "in work", there is no need for the resource adapter to participate in sync point activities. The resource adapter can avoid unnecessary processing by calling DMSCHREG and setting all the function flags OFF (except the end-of-work-unit flag, which should be left ON). When there is more work for the resource, the resource adapter can call DMSCHREG again and set the necessary flags ON.

Note: Changing the flag for the sync point function already in progress will not take effect until the next sync point. The resource adapter will continue to be called for the sync point function in progress, if it is registered for that function, until the sync point completes.

For complete information about the format and content of the DMSCHREG routine, see the *z/VM: CMS Callable Services Reference*.

Unregistering the Resource

As part of end-of-work-unit processing, the resource adapter must end the registration by calling the DMSUNREG (Unregistration) CSL routine. The SPM will then remove the entry for this instance of registration from its coordination list.

Note: Remember that the registration represents a resource and resource adapter pair for a specific work unit. The same resource adapter could be registered with a different resource on the same work unit or on a different work unit.

Unregistration takes effect immediately. Do not call this routine from any exit except postcoordination while a sync point is in progress. Otherwise, the resource adapter will not be driven for any subsequent processing by the SPM.

For information about the format and content of DMSUNREG, see the *z/VM: CMS Callable Services Reference*.

CRR Exits to Registered Resource Adapters

A user application *explicitly* starts CRR processing when the application issues a commit or backout (rollback) request:

- SAA resource recovery (also known as CPI resource recovery) routines:
  - SRRCMIT
  - SRRBACK.
- z/VM resource recovery CSL routines:
  - DMSCOMM
  - DMSROLLB.
- Resource manager verbs.

  Note: If the application issues a resource manager commit or backout verb, the resource adapter for that resource manager must translate the request into a CRR commit or rollback.

When the SPM is executing one of the CRR sync point functions, it calls the exit routines of the resource adapters registered for that function for the CMS work unit being processed.

A user application *implicitly* starts CRR processing when it:

- Reaches CMS end-of-command processing.

  When CMS reaches normal end of command, end-of-command processing issues a CRR commit for all active CMS work units. After this is done, end-of-command processing calls exits to the resource adapters registered for CRR's end-of-work-unit function.
- Calls DMSRETWU (Return Work Unit ID) processing.

  DMSRETWU issues a CRR commit for the work unit being returned. After this is done, DMSRETWU calls exits to the resource adapters registered for CRR's end-of-work-unit function.
- Reaches CMS end-of-subset processing.

  When CMS reaches normal end of subset, end-of-subset processing issues a CRR commit for all CMS work units obtained during subset mode. After this is done, end-of-subset processing calls exits to the resource adapters registered for CRR's end-of-work-unit function.
- Calls DMSPURWU (Purge Work Unit IDs) processing.

  When an application requests CMS to purge work unit IDs, CMS issues a CRR rollback (backout) for all CMS work units. After this is done, DMSPURWU calls exits to the resource adapters registered for CRR's end-of-work-unit function.
- Reaches CMS abend processing.

  When an application abends, CMS issues a CRR rollback (backout) for all CMS work units. After this is done, abend processing calls exits to the resource adapters registered for CRR's end-of-work-unit function.

Synchronous and Asynchronous Exit Processing

The resource adapter exit interface can be used in either a synchronous or asynchronous manner. A resource adapter's exits will be entered disabled for interrupts.

Based on performance considerations, the SPM uses the request ID field in the exit call to indicate the type of processing the resource adapter should use to communicate with its resource manager. When a sync point involves a single resource adapter, synchronous processing provides better performance. The SPM indicates synchronous processing by sending a request ID value of 0. When a sync point involves

multiple resource adapters, asynchronous processing provides better performance. The SPM indicates asynchronous processing by sending a nonzero request ID.

It is a requirement that adapter exits *do not* enable for interrupts as part of their processing, but instead use the asynchronous capability provided by CRR. The SPM will enable for interrupts in a carefully controlled manner.

Synchronous Processing

A resource adapter must use the WAIT=YES parameter on any APPC/VM communications operating synchronously in a resource adapter exit. However, operating in this fashion could have a serious impact on the performance of the sync point if there is more than one resource participating in the sync point.

Resource adapter exits for the postcoordination and backout-required functions must operate only synchronously.

In synchronous processing, the resource adapter's exit routine must set one of the following return codes.

RC
Meaning

0
ADACOMP (Complete). The exit has completed its activity. The response code is an output parameter of exit processing. The error block and actual error data length (if your resource manager supports them) should be updated if errors or warnings occurred.

8
ADACOMPD (Complete—Default Response). The exit did not recognize the action being driven by the SPM. The SPM should not look at the response code parameter, but instead should assume the default response.

12
ADARCAF (Adapter Failure). Exit processing had a failure. This code should be used for such things as running out of virtual storage. The SPM assumes a response code value of ADAAERR (Adapter Failure).

An example of a resource adapter using the adapter exit interface synchronously is when all resource processing is done within a virtual machine and there is no communication outside of the virtual machine.

Asynchronous Processing

A resource adapter should use the WAIT=NO parameter on any APPC/VM communications operating asynchronously in a resource adapter exit. Because CPI Communications routines operate synchronously, using these routines in a resource adapter exit could have a serious impact on performance.

In asynchronous processing, the resource adapter's exit routine must set one of the following return codes.

RC
Meaning

0
ADACOMP (Complete). The exit has completed its activity. The response code is an output parameter of exit processing. The error block and actual error data length (if your resource manager supports them) should be updated if errors or warnings occurred.

4
ADAREDRV (Redrive). The exit started the action, but the exit must be redriven when some asynchronous activity is complete, because more processing is needed to determine the correct response.

The resource adapter's interrupt handler must call the Mark Request ID (DMSMARK) CSL routine, passing the request ID that the resource adapter received from the SPM in the exit call. If the interrupt handler is able to determine what the response is, it can pass the response back to the SPM in the *optional_data_word_1* parameter on DMSMARK. If this parameter contains a nonzero value, the SPM

will not redrive the exit. Otherwise, the exit will supply a response code when it completes processing after being redriven.

It is a good idea to take advantage of the *optional_data_word_1* parameter because the broadcast will be faster. However, a resource adapter should not add exorbitant path length to its interrupt handler if the response cannot be determined quickly.

If errors and warnings occurred, the error block should be updated directly by the interrupt handler (again, only if it is able to know what the errors are), and the new actual error data length should be passed to DMSMARK in the *optional_data_word_2* parameter.

For complete information about the format and content of the DMSMARK routine, see the *z/VM: CMS Callable Services Reference*.

8
ADACOMPD (Complete—Default Response). The exit did not recognize the action being driven by the SPM. The SPM should ignore the response code parameter, and instead should assume the default response.

12
ADARCAF (Adapter Failure). The exit processed but had a failure. This code should be used for such things as running out of virtual storage. The SPM assumes a response code value of ADAAERR (Adapter Failure).

The following sequence describes asynchronous communication between a resource adapter and its resource manager:

1. The resource adapter saves the request ID passed by the resource manager in a place where its interrupt handler can find it.
2. The resource adapter's exit routine sets the ADAREDRV (Redrive) return code to be returned to the SPM.
3. If any participants have asked to be redriven, the SPM enables for interrupts using the CSL routine DMSCWAIT. Otherwise, the SPM continues with the next sync point action.
4. When the resource adapter's resource manager responds to the action, the interrupt is presented by CP.
5. When the asynchronous event completes, the resource adapter's interrupt handler gets control and uses the DMSMARK routine to pass the saved request ID to the SPM.
6. The SPM awakens and redrives all resource adapters that set ADAREDRV and did not use the *optional_data_word_1* parameter when calling DMSMARK.
7. If the resource adapter has not finished processing, it can set ADAREDRV again to be returned to the SPM.
8. Go back to step "3" on page 263.

CRR's Multitasking Dispatcher Exit

CRR provides a way for multitasking dispatchers (servers) to use the asynchronous exit from CRR. You must replace the CMS-supplied wait routine, DMSCWAIT, with your own wait routine. This allows the multitasking dispatcher to process other work while CRR waits for resource adapters, on behalf of their resource managers involved in CRR processing, to complete their work. See Chapter 17, "Writing a CRR Wait Routine for Multiuser Server Applications," on page 251.

Writing Resource Adapter Exit Routines

You must write a CSL routine to handle exit calls from the SPM to your resource adapter. This section describes the exit routine parameters and discusses the type of processing your exit routine must do for each of the possible sync point function and action calls from the SPM. For general information about writing a CSL routine, see the *z/VM: CMS Application Development Guide for Assembler*. For information about the CSLENTRY, CSLGETP, and CSLEXIT macros used in writing a CSL routine, see the *z/VM: CMS Macros and Functions Reference*.

Because the backout-required exit is driven from the interrupt handler, the processing allowed in that exit is very restricted. (See "Exit Routine Processing" on page 266.) Therefore, you may write a separate CSL routine to handle the backout-required exit.

You identify the names of your exit routines to the SPM when you register your resource for CRR participation. See "Registering a Resource for CRR" on page 257.

Note: The exit routines must be loaded into the application virtual machine (using the RTNLOAD command) before the resource adapter can use the DMSREG routine to register for CRR participation.

Exit Routine Parameters

The IBM-supplied CSL template file for the resource adapter exit routine is ADAPTERX TEMPLATE, shown in Figure 43 on page 264. This file identifies the parameters the SPM supplies on the exit call and the parameters your exit routine must supply as output to the SPM. Even if you plan to write a separate CSL routine for the backout-required exit, all of the exit functions use the same parameters; therefore, you can use ADAPTERX TEMPLATE for both exit routines. For information about the general format of a CSL template file, see the *z/VM: CMS Application Development Guide for Assembler*.

IBM supplies the ADAPTRC macro (in the DMSGPI MACLIB) to define the constants (sync point functions, sync point actions, return codes, and response codes) used in the resource adapter exit routines. The name of each constant begins with the letters ADA. For information about the contents of this macro, see the *z/VM: CMS Macros and Functions Reference*.

```
10  10                  10 Parms Maximum, 10 Required
SBIN  4  OUTPUT         Return Code
SBIN  4  OUTPUT         Response Code
SBIN  4  INPUT          Function
SBIN  4  INPUT          Action
SBIN  4  INPUT          Adapter Token
SBIN  4  INPUT          Request ID
FCHR  0  INOUT          Error Block
SBIN  4  INOUT          Usable Error Data Length
FCHR  0  INPUT          LUWID
FCHR  0  INPUT          Transaction Tag
```

Figure 43. ADAPTERX TEMPLATE File

ADAPTERX TEMPLATE contains templates for 10 required parameters:

1. The ***return code*** from the exit routine. Your routine must return one of the following codes:

   Table 32. Resource Adapter Exit Routine Return Codes

ADAPTRC Constant	RC	Description and SPM Action
ADACOMP (Complete)	0	The exit routine has completed. The SPM checks the response code for the results of driving the exit.
ADAREDRV (Redrive)	4	The exit routine has started processing, and asynchronous processing continues. The SPM redrives the exit when asynchronous processing completes.
ADACOMPD (Complete—Default Response)	8	The exit routine has completed processing and could not recognize the action value. The SPM assumes the response code has the default value.

Table 32. Resource Adapter Exit Routine Return Codes (continued)

ADAPTRC Constant	RC	Description and SPM Action
ADARCAF (Adapter Failure)	12	The exit routine cannot complete processing for some reason, and the response code contains no meaningful information. The SPM assumes a response code value of ADAAERR (Adapter Failure).

2. The ***response code*** from the exit routine. The response code is meaningful only when the return code has a value of ADACOMP (Complete). There are specific response codes that your exit routine can return for each sync point function and action call. The response codes are described in the next section ("Exit Routine Processing" on page 266).
3. The sync point ***function*** for which the SPM calls the exit to the resource adapter. The SPM sets this parameter to one of the values shown in Table 33 on page 265.
4. The sync point ***action*** for which the SPM drives the exit to the resource adapter. Table 33 on page 265 lists the possible actions that the SPM can specify for each sync point function.

   Note: There are two sets of actions for the coordination exit call. The first set, listed in Table 33 on page 265, is used when a resource adapter is participating as an agent in the sync point. The second set, defined in the ADAPTRC macro but not listed here, is used when the SPM drives the resource adapter as the initiator of the sync point; that is, when the sync point involves protected conversations.

Table 33. Synchronization Point Functions and Actions

Function	Actions
ADAPRCF (Precoordination)	ADAPRCOM (Precoordination Commit) ADAPRBCK (Precoordination Backout)
ADACORF (Coordination)	ADAPREP (Prepare) ADARQCMT (Request Commit) ADACMTD (Committed) ADACMTDL (Committed with New LUWID) ADANEWL (New LUWID) ADABOUT (Backout) ADABOUT2 (Second Phase Backout) ADAOKBO (OK Backout) ADAPTRS (Prepare to Resynchronize) ADADA (Deallocate Abend) ADAIOKBO (Initiator OK Backout)
ADAPSCF (Postcoordination)	ADAPSCOM (Postcoordination Commit) ADAPSBCK (Postcoordination Backout) ADAPSSC (Postcoordination State Check) ADAPSABN (Postcoordination Abnormal Termination)
ADAEWUF (End of Work Unit)	ADAEWPUR (Purge Work Unit) ADAEWRET (Return Work Unit) ADAEWEOC (End of Command) ADAEWABN (Abend) ADAEWSS (End of CMS Subset)

Table 33. Synchronization Point Functions and Actions (continued)

Function	Actions
ADABORQF (Backout Required)	ADABRQBO (Backout) ADABRQRF (Resource Failure) ADABRQDA (Deallocate Abend)

5. The ***adapter token*** that your resource adapter uses to identify the resource, in case the resource adapter is handling multiple resources. You supply this value to the SPM when you register the resource for the work unit, and the SPM passes the value back in the exit call.
6. The ***request ID*** that the SPM associates with the adapter token. The SPM passes this value to the resource adapter to support asynchronous processing. If the value passed is 0, the resource adapter must use synchronous processing. See "Synchronous and Asynchronous Exit Processing" on page 261.
7. The ***error block*** allocated for this resource. You determine the size of the buffer you want to use. You supply this value to the SPM when you register the resource.

   Note: To have a usable error buffer, you must specify a value greater than 4. The first four bytes of the buffer are used to specify the length of the buffer, so a 4-byte buffer has no room for any error data. If you specify a value of 0, the SPM passes a default 4-byte buffer to the resource adapter on the exit call. Values of 1, 2, or 3 are not valid.
8. The ***actual error data length*** is the amount of usable information in the error block. The SPM sets this parameter to 4 at the start of a sync point. Until the start of the next sync point, the resource adapter must manage this parameter to indicate how much of the error block is in use. See "Detailed Error Passback Support" on page 285.
9. The SNA LU 6.2 architected logical unit of work identifier (***LUWID***). The range for this variable is 0-26 characters.
   - In the ADAPRCF (Precoordination Function) call, this is the LUWID to be used with the current sync point. If no LUWID is specified in the exit call, the SPM passes the LUWID as part of the first coordination phase.
   - In the ADACORF (Coordination Function) call, this is normally the LUWID to used with the current sync point. However, in some cases (the ADACMTDL and ADANEWL actions, for example) this is the LUWID to be used for the next sync point.
   - In the ADAPSCF (Postcoordination Function), ADAEWUF (End-of-Work-Unit Function), and ADABORQF (Backout-Required Function) calls, this is the LUWID to be used for the next sync point.
10. The ***transaction tag*** for this sync point (work unit), if the application has assigned one. For more information about transaction tags, see "Setting Up to Ensure Data Integrity" on page 243 and the description of the DMSSETAG CSL routine in the *z/VM: CMS Callable Services Reference*.

Exit Routine Processing

The execution attributes of a resource adapter exit are:

- Disabled for interrupts
- Supervisor state in Program Status Word (PSW)
- PSW key zero (can access system key or user key data)
- AMODE 31 (can be used in an XC-mode virtual machine).

There are certain restrictions on the type of processing that the exit routine is permitted to do:

- The exit routine may not enable itself for interrupts.
- No SVCs may be issued (applies to all exit functions except end of work unit).
- No asynchronous activity is allowed (applies only to the backout-required exit).

- No communications such as APPC communications can take place (applies only to the backout-required exit).

The SPM passes the following information to the resource adapter on the exit call:

- Sync point function being called
- Action to be performed by the resource adapter
- Adapter token to identify the resource
- Request ID to be used when doing asynchronous processing
- Address of the buffer where detailed error information can be stored for the application
- Logical unit of work identifier (LUWID)
- Transaction tag to identify the sync point.

The resource adapter exit routine that you write must:

- Perform any requested action in the resource for each sync point function you intend to register for
- Pass the SPM-supplied LUWID to the resource manager
- Complete any other processing you want to do for the sync point function and action
- Return architected CRR return codes and response codes and optional error information to the SPM.

Specific exit processing considerations for each sync point function and action are described in the following sections.

Normally, the resource adapter and resource manager maintain a continuous conversation from the time an application first accesses the resource until the piece of work is completed. However, there are times when a resource adapter should break its connection to the resource manager prematurely, such as when an adapter exit is driven for the ADAPTRS (Prepare to Resynchronize) action. At that time, the resource adapter should deallocate the conversation to prevent any further updates to the resource. Also, if the resource adapter responds to an exit call from the SPM with a response such as ADAPV (Protocol Violation), the SPM assumes that communications between the resource adapter and the resource manager has already ended.

ADAPRCF (Precoordination Function) Exit

The possible actions in the precoordination exit are:

- ADAPRCOM (Precoordination Commit)
- ADAPRBCK (Precoordination Backout).

The precoordination exit gives your resource adapter a chance to do other processing before the SPM calls the registered resources for the actual commit or backout processing. The resource adapter can use this exit to do such things as flush buffers and release read locks. A very important task in this exit is for the resource adapter to verify that it is in the proper state for a sync point and then inform the SPM. Resource adapters are driven for the precoordination exit before any protected conversations are driven.

This exit provides optional asynchronous capability by using the ADAREDRV (Redrive) return code. However, if the SPM passes a request ID of 0 on the call, the resource adapter must use synchronous processing.

ATTENTION

This exit is intended to provide an opportunity for the resource adapter to communicate with the resource manager to ensure that everything is ready for the sync point. It should not be used to do any work with another resource. Except for protected conversations, which are always called last, you cannot predict the order in which the SPM will call the resource adapters registered for the work unit; therefore, you cannot predict when those changes will be committed.

SFS flushes its buffers *before* the precoordination exit. Do not use this exit to write to any SFS file pool, because the changes will not be committed.

ADAPRCOM (Precoordination Commit) Action

This action occurs before coordination exit processing. The application program has requested a commit sync point.

Table 34 on page 268 lists (in order of increasing severity) the response codes your resource adapter can return for an ADAPRCOM action call, and the resulting action taken by the SPM for each response.

Table 34. Resource Adapter Exit Routine Response Codes for the ADAPRCOM (Precoordination Commit) Sync Point Action Call

Response Code	Macro Value	Resource's Reason for Issuing This Response	Action Taken by the SPM
ADAOK	100	Ready to start a commit. Default response.	Performs coordination exit processing for registered adapters for commit sequence starting with the action of ADAPREP or ADARQCMT.
ADACSCHK	76	Adapter cannot commit due to its application state.	1. Bypasses all coordination exit processing. 2. Performs postcoordination exit processing for registered adapters with an action value of ADAPSSC (State Check).
ADACPERR	84	Adapter cannot commit due to condition other than application state.	Performs coordination exit processing for registered adapters for backout starting with the action of ADABOUT.
ADARF	8	Resource unavailable due to session failure or resource manager failure.	Performs coordination exit processing for registered adapters for backout starting with the action of ADABOUT.
ADAAERR	212	Adapter failed; resource unavailable.	Performs coordination exit processing for registered adapters for backout starting with the action of ADABOUT.

ADAPRBCK (Precoordination Backout) Action

This action occurs before coordination exit processing. The application program has requested a backout sync point.

Table 35 on page 268 lists (in order of increasing severity) the response codes your resource adapter can return for an ADAPRBCK action call, and the resulting action taken by the SPM for each response.

Table 35. Resource Adapter Exit Routine Response Codes for the ADAPRBCK (Precoordination Backout) Sync Point Action Call

Response Code	Macro Value	Resource's Reason for Issuing This Response	Action Taken by the SPM
ADAOK	100	Ready for backout. Default response.	Performs coordination exit processing for registered adapters for backout starting with the action of ADABOUT or ADADA.
ADABSCHK	80	Adapter cannot back out resource due to its application state. (Participating resource adapters should try to avoid using this response code.)	1. Bypasses all coordination processing; no backout. 2. Performs postcoordination exit processing for registered adapterswith an action value of ADAPSSC (State Check).

Table 35. Resource Adapter Exit Routine Response Codes for the ADAPRBCK (Precoordination Backout) Sync Point Action Call (continued)

Response Code	Macro Value	Resource's Reason for Issuing This Response	Action Taken by the SPM
ADARF	8	Resource unavailable due to session failure or resource manager failure.	Performs coordination exit processing for registered adapters for backout starting with the action of ADABOUT or ADADA.
ADAAERR	212	Resource unavailable due to adapter failure.	Performs coordination exit processing for registered adapters for backout starting with the action of ADABOUT or ADADA.

ADACORF (Coordination Function) Exit

The possible actions in the coordination exit are:

- ADAPREP (Prepare)
- ADARQCMT (Request Commit)
- ADACMTD (Committed)
- ADACMTDL (Committed with New LUWID)
- ADANEWL (New LUWID)
- ADABOUT (Backout)
- ADABOUT2 (Second Phase Backout)
- ADAOKBO (OK Backout)
- ADAPTRS (Prepare to Resynchronize)
- ADADA (Deallocate Abend)
- ADAIOKBO (Initiator OK Backout).

The SPM optimizes sync point processing for a work unit, if possible. After all precoordination exits have completed, the SPM looks at all resources registered for the work unit to determine if a simple commit can be done. The SPM looks at three flags in each resource adapter's registration:

coordination function flag
If this flag is ON, there is work to be coordinated for this resource; the resource adapter needs to be involved in sync point processing.

simple-commit flag
If this flag is ON, this resource adapter and its resource manager understand a simple (one-phase) commit. No logging is done for a simple commit, because no resynchronization activity is needed if a simple commit fails.

write-mode flag
If this flag is OFF, the work in progress does not involve updates to this resource. For a simple commit, only one resource can be registered as write-mode on the work unit.

The resource adapter sets these flags when it registers for CRR participation. (See “Registering a Resource for CRR” on page 257.) The simple-commit flag does not indicate that a simple commit *should* be done, but rather that the resource adapter can support a simple commit *if* one is done. If there is a need for changing this flag, the resource adapter must unregister and reregister.

To ensure consistency, changes made to a resource's registration while the coordination stage of a sync point is in progress will not take effect until the coordination stage completes. An exception to this is the unregistration (DMSUNREG) routine, which takes effect immediately. Other changes made during coordination will take effect prior to the adapter exits being driven for postcoordination.

The SPM also optimizes a two-phase commit. No logging is done for those resources involved in the sync point that are read-mode (write-mode flag set OFF) and understand a simple commit (simple-commit flag set ON).

This exit provides optional asynchronous capability by using the ADAREDRV (Redrive) return code. However, if SFS passes a request ID of 0 on the call, the resource adapter must synchronous processing.

Coordination Logging

All sync point logging done by CRR is handled during the coordination function of a two-phase commit. If the application issues a backout verb, the participating resource manager is not required to do any logging. The resource manager is required (by CRR) to do logging only during a two-phase commit.

For a normal two-phase commit, the participating resource manager must do these logging operations:

1. Logging agent, prepared

   During the ADAPREP action, the resource manager must log the LUWID associated with the commit. It is also recommended that the transaction tag be logged. This information must be logged before the resource responds positively to ADAPREP.
2. Erasing the log

   The resource manager can erase the log data from step 1 after it has successfully processed the ADACMTD or ADACMTDL action.

   Note: It is possible that the ADAPREP action can be followed by a backout. In that case, the resource manager can erase the log data after the changes are backed out.

After "logging agent, prepared" has been logged, there are two states that can be logged as part of error processing. These states represent action taken without receiving direction from CRR:

- Heuristic backout

  The resource manager or an operator caused the changes to be backed out. This state could cause a problem because other participants in the sync point might have committed their changes.
- Heuristic committed

  The resource manager or an operator caused the changes to be committed. This state could cause a problem because other participants in the sync point might have backed out their changes.

These two states should only occur if there is a long delay in the sync point, or if the communication path to the resource adapter is lost. These states will normally be cleaned up by resynchronization recovery. When the resynchronization recovery flow arrives, the resource manager can determine if the heuristic action caused an out-of-sync situation. If so, the resource manager should report it to the operator. If not, the resource manager should erase the log record according to normal resynchronization recovery logic.

The resource manager must supply facilities that the operator can use to resolve any problems in the resource manager's log. See “Resource Manager Resynchronization Facilities” on page 286.

Break Tree Processing

The protected conversation tree (more commonly called the allocation tree) describes the structure of nodes that are allocating protected conversations within a coordinated transaction. See the description of allocation and sync point trees in the *z/VM: CMS File Pool Planning, Administration, and Operation*. Break tree processing is done to ensure the integrity of the LUWID whenever a protected conversation abnormally ends. An LUWID is associated with each sync point, and it is an SNA LU 6.2 requirement that LUWIDs be unique within the SNA network.

If any error breaks a link in the allocation tree, then two smaller trees are created. An allocation tree can have only one LUWID. The key concept of break tree processing is to let one part of the allocation tree survive and keep the LUWID, and to break apart the other tree and give each piece a different LUWID. If the tree gets broken above you, then you are part of the sub-tree that gets dismantled. If the break occurs below you in the allocation tree, then you are part of the tree that lives.

Figure 44 on page 271 shows an allocation tree in which VMA talks to VMB, VMB talks to VMC and VMD, and VMD talks to VME.

- If the protected conversation between VMA and VMB has an error and ends, then break tree processing requires that all the other protected conversations end.
- If the protected conversation between VMD and VME has an error and ends, then break tree processing requires that no other protected conversations end.
- If the protected conversation between VMB and VMD has an error and ends, then break tree processing requires that the protected conversation between VMD and VME also ends.

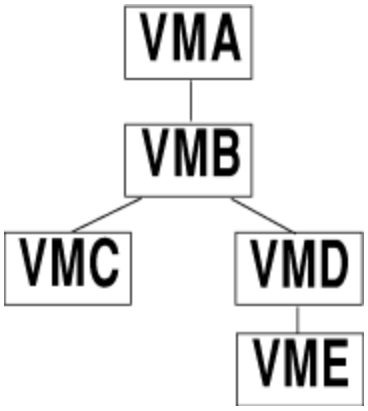


Figure 44. Break Tree Processing

Break tree processing, if it occurs, is part of SPM coordination. The SPM accomplishes the break tree processing by sending an ADADA (Deallocate Abend) action to all resource adapters registered for the work unit. For most resource adapters, this deallocate will be a null operation. If applicable, the resource adapter should make sure that the resource manager is aware that a new LUWID will be used for the next sync point. The new LUWID will *not* be created by simply incrementing the sequence number field.

Extra Backout

In some situations, the SNA LU 6.2 architecture requires that an extra backout be done following an attempted commit before returning to an application. This extra backout is done to make sure that SPMs and resource managers in two separate allocation trees do not unwittingly write log records using the same LUWID.

An LUWID is associated with each sync point. It is an SNA LU 6.2 requirement that LUWIDs be unique within the SNA network. Normally, when a session outage occurs, one part of the tree keeps the LUWID, and break tree processing occurs for the other part of the tree, as described above. However, there are a few cases when a session outage is not immediately detected by part of the tree, such as when the outage occurs on the forget flow to the initiator of the sync point.

If the session outage is not detected, resource managers might write log information using the same LUWID for two separate allocation trees. Usually, the session outage is discovered during the next sync point, reported as a resource failure, and the sync point becomes a backout. However, if another session outage occurs during this second sync point, it is possible that two now-separate branches of a tree could attempt resynchronization of unrelated work using the same LUWID.

To avoid this situation, an extra backout is done in the tree that owns the LUWID when some part of the tree detects the session outage. The extra backout causes the sequence number field of the LUWID to be incremented, which means that the two sub-trees have different LUWIDs.

The SPM accomplishes the extra backout by sending an ADABOUT (Backout) action to all resources registered for the work unit. For most resource adapters, this extra backout will be a null operation. If applicable, the resource adapter should make sure that the resource manager is aware of the new LUWID with the incremented sequence number field.

ADAPREP (Prepare) Action

This action is the start of a two-phase commit. The resource adapter should tell its resource manager to put itself in a state from which it can either commit or back out the changes. The resource manager must write the LUWID and other recovery information in its log.

Table 36 on page 272 lists (in order of increasing severity) the response codes your resource adapter can return for an ADAPREP action call, and the resulting action taken by the SPM for each response.

Table 36. Resource Adapter Exit Routine Response Codes for the ADAPREP (Prepare) Sync Point Action Call

Response Code	Macro Value	Resource's Reason for Issuing This Response	Action Taken by the SPM
ADARQCMT	28	Prepare has been done. Default response code.	Continue with second phase of commit with the action of ADACMTD or ADACMTDL.
ADAFGET	60	Resource manager has successfully completed work for this sync point. This response should be used only when both commit and backout would leave the resource in the same state (for example, if no updates have been made to it).	Continue with second phase of commit for other resources. Bypass this resource unless the action of ADANEWL is required to send a new LUWID to the resource.
ADABOUT	4	Unable to commit changes, changes have been backed out.	Back out all other resources, then redrive this resource with the action of ADAOKBO to acknowledge the backout.
ADABORIP	68	Unable to commit changes, changes have been backed out. Resynchronization is in progress for some resource.	Back out all other resources, then redrive this resource with the action of ADAOKBO to acknowledge the backout.
ADAPERR	208	Recoverable application error. Changes have neither been prepared nor backed out.	Back out all resources with the action of ADABOUT2.
ADAHMIX	88	One or more protected resources have committed due to a resource manager or system administrator decision.	Back out all other resources with the action of ADABOUT2.
ADADA	20	The resource manager has backed out the changes, and is unavailable for further work.	Back out all other resources with the action of ADABOUT2. Perform break tree or extra backout processing.
ADARF	8	The resource manager failed, and did not give information about the state of the changes.	Back out all resources with the action of ADABOUT2.
ADAPV	92	The adapter received an invalid response code from its resource manager and the link has been severed.	Back out all other resources with the action of ADABOUT2. Perform break tree or extra backout processing.
ADAAERR	212	The adapter failed, and did not give information about the state of the resource or the changes.	Back out all resources with the action of ADABOUT2.

ADARQCMT (Request Commit) Action

This action is used if the SPM determines that it is possible to perform a simple commit instead of a two-phase commit. The resource manager should commit the changes. Because a simple commit is a one-phase operation, for performance reasons it is preferred whenever possible. For example, a simple commit might be performed when only one resource is registered for a work unit.

Table 37 on page 273 lists (in order of increasing severity) the response codes your resource adapter can return for an ADARQCMT action call, and the resulting action taken by the SPM for each response.

Table 37. Resource Adapter Exit Routine Response Codes for the ADARQCMT (Request Commit) Sync Point Action Call

Response Code	Macro Value	Resource's Reason for Issuing This Response	Action Taken by the SPM
ADACMTD	56	Changes have been committed. Default response.	End of processing for this resource. Commit all read only resources with the action of ADACMTD.
ADABOUT	4	Changes have been backed out.	End of processing for this resource. Back out all read only resources with the action of ADABOUT.
ADAPERR	208	Changes have neither been committed nor backed out.	Back out all the read only resources with the action of ADABOUT.
ADARF	8	The resource manager failed, and did not give information about the state of the changes.	Back out all read only resources with the action of ADABOUT and redrive this resource adapter with ADAPTRS.
ADAAERR	212	The adapter failed, and did not give information about the state of the resource or the changes.	Back out all read only resources with the action of ADABOUT and redrive this resource adapter with ADAPTRS.

ADACMTD (Committed) or ADACMTDL (Committed With New LUWID) Action

This action is the start of the second phase of a two-phase commit, when the second phase is a commit. The resource should commit all changes and erase the log data when done. Most resource adapters do not act differently for the two types of committed action. The difference is that for the ADACMTDL action, the LUWID passed to the adapter exit is the one that will be used for the next sync point.

Table 38 on page 273 lists (in order of increasing severity) the response codes your resource adapter can return for an ADACMTD or ADACMTDL action call, and the resulting action taken by the SPM for each response.

Table 38. Resource Adapter Exit Routine Response Codes for the ADACMTD (Committed) or ADACMTDL (Committed with New LUWID) Action Call

Response Code	Macro Value	Resource's Reason for Issuing This Response	Action Taken by the SPM
ADAFGET	60	Changes have been committed. Default response.	End of processing.
ADAFRIP	64	Resynchronization is in progress for some resources. Changes will be committed.	End of processing.
ADAHMIX	88	One or more protected resources have backed out due to a resource manager or system administrator decision.	Redrive this resource adapter with ADAPTRS.

Table 38. Resource Adapter Exit Routine Response Codes for the ADACMTD (Committed) or ADACMTDL (Committed with New LUWID) Action Call (continued)

Response Code	Macro Value	Resource's Reason for Issuing This Response	Action Taken by the SPM
ADARF	8	The resource manager failed, and did not give information about the state of the changes.	Redrive this resource adapter with ADAPTRS.
ADAPV	92	The adapter received an invalid response code from its resource manager and the link has been severed. One or more protected resources may have backed out due to a protocal violation.	Report the protocol violation to the recovery server. Perform break tree or extra backout processing.
ADAPVPRT	48	The adapter received the protocol violation indication from its conversation partner and the link has been severed.	Report the protocol violation to the recovery server. Perform break tree or extra backout processing.
ADAAERR	212	The adapter failed, and did not give information about the state of the resource or the changes.	Redrive this resource adapter with ADAPTRS.

ADANEWL (New LUWID) Action

This action is used if a resource adapter responds ADAFGET (Forget) when driven for the ADAPREP (Prepare) action, and a new LUWID is to be used for the next sync point. This LUWID is not the current LUWID that has been incremented, but is newly created. The new LUWID is passed as input.

Table 39 on page 274 lists (in order of increasing severity) the response codes your resource adapter can return for an ADANEWL action call, and the resulting action taken by the SPM for each response.

Table 39. Resource Adapter Exit Routine Response Codes for the ADANEWL (New LUWID) Sync Point Action Call

Response Code	Macro Value	Resource's Reason for Issuing This Response	Action Taken by the SPM
ADAOK	100	LUWID has been passed to the resource manager if necessary. Default response.	End of processing.
ADAPERR	208	A send error occurred, and the resource manager did not receive the new LUWID.	Redrive this resource adapter with ADAPTRS.
ADARF	8	The resource manager failed, and did not receive the new LUWID.	Redrive this resource adapter with ADAPTRS.
ADAPV	92	The adapter received an invalid response code from its resource manager and the link has been severed.	Perform break tree or extra backout processing.
ADAAERR	212	The adapter failed, and did not receive the new LUWID.	Redrive this resource adapter with ADAPTRS.

ADABOUT (Backout) Action

This action tells the resource adapter that it is to back out all changes, as a result of a backout being issued by the application program, extra backout processing, or errors during precoordination commit.

Table 40 on page 275 lists (in order of increasing severity) the response codes your resource adapter can return for an ADABOUT action call, and the resulting action taken by the SPM for each response.

Table 40. Resource Adapter Exit Routine Response Codes for the ADABOUT (Backout) Sync Point Action Call

Response Code	Macro Value	Resource's Reason for Issuing This Response	Action Taken by the SPM
ADAOKBO	72	Changes have been backed out. Default response.	End of processing.
ADABOUT	4	This resource independently started a backout and should be treated as the initiator of the backout.	Send ADAIOKBO to this resource adapter to acknowledge the backout.
ADAAWARN	216	Resource adapter warning.	Continue backout processing.
ADAPERR	208	Application error. Changes have neither been committed nor backed out.	Redrive this resource adapter with ADAPTRS.
ADADA	20	The resource manager has backed out the changes, and is unavailable for further work.	Perform break tree or extra backout processing.
ADARF	8	The resource manager failed, and did not give any information about the state of the changes.	Redrive this resource adapter with ADAPTRS.
ADAPV	92	The adapter received an invalid response code from its resource manager and the link has been severed.	Because the resource is not in doubt, it is assumed that the changes will be backed out. Perform break tree or extra backout processing.
ADAAERR	212	The adapter failed, and did not give information about the state of the resource or the changes.	Redrive this resource adapter with ADAPTRS.

ADABOUT2 (Second Phase Backout) Action

This action is the start if the second phase of a two-phase commit, when the second phase is a backout. The resource should back out all changes and erase the log data when done. The resource has already been asked to process an ADAPREP (Prepare) action, but at least one resource voted "backout" during the first phase of the two-phase commit.

Table 41 on page 275 lists (in order of increasing severity) the response codes your resource adapter can return for an ADABOUT2 action call, and the resulting action taken by the SPM for each response.

Table 41. Resource Adapter Exit Routine Response Codes for the ADABOUT2 (Second Phase Backout) Sync Point Action Call

Response Code	Macro Value	Resource's Reason for Issuing This Response	Action Taken by the SPM
ADAOKBO	72	Changes have been backed out. Default response.	End of processing.
ADABORIP	68	Unable to back out changes, changes will be backed out. Resynchronization is in progress for some resource.	End of processing

Table 41. Resource Adapter Exit Routine Response Codes for the ADABOUT2 (Second Phase Backout) Sync Point Action Call (continued)

Response Code	Macro Value	Resource's Reason for Issuing This Response	Action Taken by the SPM
ADAHMIX	88	The resource manager committed the changes.	Redrive this resource adapter with ADAPTRS.
ADARF	8	The resource manager failed, and did not give any information about the state of the changes.	Redrive this resource adapter with ADAPTRS.
ADAPV	92	The adapter received an invalid response code from its resource manager and the link has been severed.	Notify the recovery server of the violation for this resource. Report the damage to the recovery server operator. Perform break tree or extra backout processing.
ADAAERR	212	The adapter failed, and did not give information about the state of the resource or the changes.	Redrive this resource adapter with ADAPTRS.

ADAOKBO (OK Backout) Action

This action tells the resource adapter that the SPM has received the ADABOUT (Backout) response code from the resource adapter for a previous ADAPREP (Prepare) action request (see "ADAPREP (Prepare) Action" on page 272), and that the SPM has backed out its resources.

Table 42 on page 276 lists (in order of increasing severity) the response codes your resource adapter can return for an ADAOKBO action call, and the resulting action taken by the SPM for each response.

Table 42. Resource Adapter Exit Routine Response Codes for the ADAOKBO (OK Backout) Sync Point Action Call

Response Code	Macro Value	Resource's Reason for Issuing This Response	Action Taken by the SPM
ADAOK	100	Action has been sent without error. Default response.	End of processing.
ADAPV	92	The adapter received an invalid response code from its resource manager and the link has been severed.	Perform break tree or extra backout processing.
ADAAERR	212	Unable to send ADAOK to the resource manager.	Redrive this resource adapter with ADAPTRS.

ADAPTRS (Prepare to Resynchronize) Action

This action tells the resource adapter that the SPM has detected an error which requires action that is beyond the scope of the SPM. Most of these errors will be handled by resynchronization; however, some errors will have to be handled by the resource manager.

The resource adapter, upon receiving this action, must do whatever processing is required to ensure that the resource manager is in the proper state for resynchronization processing. One of the actions the resource adapter should do is sever the link between the application (resource adapter) and the resource manager. After this action is processed, the CRR recovery server will do whatever processing it requires to ensure that resynchronization can occur.

The resource manager should assume that if the application path is broken while in the in-doubt state, then the affected work must wait for CRR resynchronization processing to tell the resource to commit or back out. However, a resource manager might receive this action in a situation where there is no resynchronization responsibility. For example, if the resource adapter returns the response code ADARF

(Resource Failure) to an ADABOUT (Backout) action, the SPM does not know what the state of the changes are, so it issues the ADAPTRS action. In that case, because the resource manager is not in the in-doubt state when ADAPTRS breaks the path, it is the resource manager's responsibility to back out the work.

Table 43 on page 277 lists (in order of increasing severity) the response codes your resource adapter can return for an ADAPTRS action call, and the resulting action taken by the SPM for each response.

Note: Returning an ADARCAF (Adapter Failure) return code to the ADAPTRS action causes the SPM to abend CMS. If possible, the resource adapter should do the necessary processing to return the ADACOMP (Complete) return code with the ADAOK response code.

Table 43. Resource Adapter Exit Routine Response Codes for the ADAPTRS (Prepare to Resynchronize) Sync Point Action Call

Response Code	Macro Value	Resource's Reason for Issuing This Response	Action Taken by the SPM
ADAOK	100	The resource manager is ready for resynchronization. Default response.	Perform break tree or backout processing.
ADAAERR	212	The resource manager was unable to get ready for resynchronization.	ABEND CMS, therefore insuring that the link between the resource manager and the application will be severed.

ADADA (Deallocate Abend) Action

This action tells the resource adapter to back out all changes in certain error conditions. For example, this action is used if the CRR recovery server is needed during the sync point, but it is not available at the start of the sync point. In this case, ADADA will be the first action in the coordination of the sync point.

It is possible that a resource adapter could be driven for backout and then driven for deallocate abend prior to postcoordination. In this case, no action will occur for most resource adapters for the deallocate abend.

Table 44 on page 277 lists (in order of increasing severity) the response codes your resource adapter can return for an ADADA action call, and the resulting action taken by the SPM for each response.

Table 44. Resource Adapter Exit Routine Response Codes for the ADADA (Deallocate Abend) Sync Point Action Call

Response Code	Macro Value	Resource's Reason for Issuing This Response	Action Taken by the SPM
ADAOK	100	Changes have been backed out or the link has been deallocated. Default response.	End of processing.
ADAPV	92	The adapter received an invalid response code from its resource manager and the link has been severed.	Perform break tree or extra backout processing.
ADAAERR	212	The adapter failed, and did not give information about the state of the resource or the changes.	Redrive the resource adapter with ADAPTRS.

ADAIOKBO (Initiator OK Backout) Action

A resource adapter becomes an initiator when the resource adapter responds ADABOUT to the action of ADABOUT. The initiator is driven with the initiator OK backout action when all agents of this SPM have responded ADAOKBO.

Table 45 on page 278 lists (in order of increasing severity) the response codes your resource adapter can return for an ADAIOKBO action call, and the resulting action taken by the SPM for each response.

Table 45. Resource Adapter Exit Routine Response Codes for the ADAIOKBO (Initiator OK Backout) Sync Point Action Call

Response Code	Macro Value	Initiator Reason	Action Taken by SPM
ADAOK	100	ADAIOKBO sent without error. Default response.	End of the sync point.
ADARF	8	The sync point initiator failed, and did not receive the action.	Redrive the resource adapter with ADAPTRS.
ADAPV	92	The adapter received an invalid response code from its resource and the link has been severed.	If there is any resynchronization responsibility, then tell the recovery server that the sync point was backed out. Perform break tree or extra backout processing.
ADAAERR	212	The adapter failed, and could not send the ADAIOKBO.	Redrive the resource adapter with ADAPTRS.

ADAPSCF (Postcoordination Function) Exit

The possible actions in the postcoordination exit are:

- ADAPSCOM (Postcoordination Commit)
- ADAPSBCK (Postcoordination Backout)
- ADAPSSC (Postcoordination State Check)
- ADAPSABN (Postcoordination Abnormal Termination).

Postcoordination processing usually entails cleanup. You might want to use this exit to do any necessary processing before the application receives control. For example, such processing could include resetting function flags.

Postcoordination exit processing does not begin until all coordination exit processing has completed (all changes have been committed or backed out). There are error cases where postcoordination exit processing occurs after precoordination exit processing has completed, with coordination exit processing bypassed. For more information on these error cases, see the “ADAPSSC (Postcoordination State Check) Action” on page 279.

Because CMS APPC support does not allow interrupts to occur during postcoordination, the processing allowed in this exit is very restricted:

- No asynchronous activity is allowed. The request ID parameter is passed on the call, but cannot be used.
- No communications such as APPC communications can take place.
- No supervisor call instructions (SVCs) may be issued.
- The exit may not enable itself for interrupts.

ADAPSCOM (Postcoordination Commit) Action

The sync point ended in a commit, which has been completed, or is being completed by resynchronization, before the exit is called for postcoordination processing. Resource adapters may want to use the DMSCHREG (Change Registration) CSL routine at this time to update the resource registration to reflect the current state. When a commit is followed by an extra backout, the action passed during postcoordination is a commit. See “Extra Backout” on page 271 for more information on extra backouts.

Table 46 on page 279 lists (in order of increasing severity) the response codes your resource adapter can return for an ADAPSCOM action call, and the resulting action taken by the SPM for each response.

Table 46. Resource Adapter Exit Routine Response Codes for the ADAPSCOM (Postcoordination Commit) Sync Point Action Call

Response Code	Macro Value	Resource's Reason for Issuing This Response	Action Taken by the SPM
ADAOK	100	Default response.	End of processing.
ADARF	8	The resource manager is no longer available due to a session outage, deallocate abend, or failure of the resource manager.	End of processing.
ADAAERR	212	The resource manager is no longer available due to failure of the adapter.	End of processing.

ADAPSBCK (Postcoordination Backout) Action

The sync point ended in a backout, which has been completed, or is being completed by resynchronization, before the exit is called for postcoordination processing. Resource adapters may want to use the DMSCHREG (Change Registration) CSL routine at this time to update the resource registration to reflect the current state.

Table 47 on page 279 lists (in order of increasing severity) the response codes your resource adapter can return for an ADAPSBCK action call, and the resulting action taken by the SPM for each response.

Table 47. Resource Adapter Exit Routine Response Codes for the ADAPSBCK (Postcoordination Backout) Sync Point Action Call

Response Code	Macro Value	Resource's Reason for Issuing This Response	Action Taken by the SPM
ADAOK	100	Default response.	End of processing.
ADARF	8	The resource manager is no longer available due to a session outage, deallocate abend, or failure of the resource manager.	End of processing.
ADAAERR	212	The resource manager is no longer available due to failure of the adapter.	End of processing.

ADAPSSC (Postcoordination State Check) Action

This action means there was a sync point request, but, due to an error condition detected during precoordination exit processing, neither commit nor backout was performed. No coordination exit processing was performed.

Because coordination exit processing was bypassed, resource adapters may need to clean up activities done or started during precoordination exit processing that normally would have been cleaned up during coordination exit processing. The state of the work unit has not changed.

Table 48 on page 280 lists (in order of increasing severity) the response codes your resource adapter can return for an ADAPSSC action call, and the resulting action taken by the SPM for each response.

Table 48. Resource Adapter Exit Routine Response Codes for the ADAPSSC (Postcoordination State Check) Sync Point Action Call

Response Code	Macro Value	Resource's Reason for Issuing This Response	Action Taken by the SPM
ADAOK	100	Default response.	End of processing.
ADARF	8	The resource is no longer available due to a session outage, deallocate abend, or failure of the resource manager.	End of processing.
ADAAERR	212	The resource is no longer available due to failure of the adapter.	End of processing.

ADAPSABN (Postcoordination Abnormal Termination) Action

An abnormal termination occurred while the SPM was processing a sync point. If resynchronization processing is started to complete the sync point, the SPM cannot determine whether it was a commit or a backout.

Table 49 on page 280 lists (in order of increasing severity) the response codes your resource adapter can return for an ADAPSABN action call, and the resulting action taken by the SPM for each response.

Table 49. Resource Adapter Exit Routine Response Codes for the ADAPSABN (Postcoordination Abnormal Termination) Sync Point Action Call

Response Code	Macro Value	Resource's Reason for Issuing This Response	Action Taken by the SPM
ADAOK	100	Default response.	End of processing.
ADARF	8	The resource is no longer available due to a session outage, deallocate abend, or failure of the resource manager.	End of processing.
ADAAERR	212	The resource is no longer available due to failure of the adapter.	End of processing.

ADAEWUF (End-of-Work-Unit Function) Exit

The possible actions in the end-of-work-unit exit are:

- ADAEWPUR (Purge Work Unit)
- ADAEWRET (Return Work Unit)
- ADAEWEOC (End of Command)
- ADAEWABN (CMS Command Abend)
- ADAEWSS (End of CMS Subset).

The end-of-work-unit exit is driven by DMSPURWU (Purge Work Unit IDs), DMSRETWU (Return Work Unit ID), CMS end-of-command, CMS abend, and CMS end-of-subset processing to tell registered resource owners that the work unit might no longer exist. The work unit is already committed or backed out before this exit is driven. In most cases, a resource adapter should leave the end-of-work-unit function flag set ON in its registration, so the resource adapter can be called to clean up and unregister.

Cleanup might involve such things as releasing storage (control blocks) and severing paths to the resource manager. For example, depending on the cause of the end of the work unit, it may be appropriate for a resource adapter to deallocate (sever) any communication paths it uses to communicate with its resource

manager (for example, its server machine) for this work unit. Resource adapters must unregister as part of their end-of-work-unit processing. The end-of-work-unit exit could also be used for accounting or performance monitoring.

This exit provides optional asynchronous capability by using the ADAREDRV (Redrive) return code. However, if SFS passes a resource ID of 0 on the call, the resource adapter must use synchronous processing.

ADAEWPUR (Purge Work Unit) Action

The SPM drives the resource adapter for this action because the application called the DMSPURWU (Purge Work Unit IDs) CSL routine to return all work unit IDs to CMS. For more information about DMSPURWU, see the *z/VM: CMS Callable Services Reference*.

If your resource adapter has any external communication paths (such as APPC/VM connections) associated with the work unit, the resource adapter should deallocate (sever) them. Failure to do so could cause a security problem. Subsequent applications, particularly in a batch machine, might not be authorized to use the communication paths, but might be able to do so if authorization checking is done only when the communication is established.

Table 50 on page 281 lists (in order of increasing severity) the response codes your resource adapter can return for an ADAEWPUR action call, and the resulting action taken by the SPM for each response.

Table 50. Resource Adapter Exit Routine Response Codes for the ADAEWPUR (Purge Work Unit) Sync Point Action Call

Response Code	Macro Value	Resource's Reason for Issuing This Response	Action Taken by the SPM
ADAOK	100	Default response.	Work unit is ended.
ADARF	8	Resource no longer available due to session outage, deallocate (abend), or failure of the resource manager.	Work unit is ended.
ADAAERR	212	Resource no longer available due to failure of the adapter.	Work unit is ended.

ADAEWRET (Return Work Unit) Action

The SPM drives the resource adapter for this action because the application called the DMSRETWU (Return Work Unit ID) CSL routine, indicating that the application has completed all work on the work unit. For more information about DMSRETWU, see the *z/VM: CMS Callable Services Reference*.

CMS return-work-unit processing for CRR includes:

- Closing all work unit SFS files and directories
- Committing the work unit.

Table 51 on page 281 lists (in order of increasing severity) the response codes your resource adapter can return for an ADAEWRET action call, and the resulting action taken by the SPM for each response.

Table 51. Resource Adapter Exit Routine Response Codes for the ADAEWRET (Return Work Unit) Sync Point Action Call

Response Code	Macro Value	Resource's Reason for Issuing This Response	Action Taken by the SPM
ADAOK	100	Default response.	Work unit is ended.

Table 51. Resource Adapter Exit Routine Response Codes for the ADAEWRET (Return Work Unit) Sync Point Action Call (continued)

Response Code	Macro Value	Resource's Reason for Issuing This Response	Action Taken by the SPM
ADARF	8	Resource no longer available due to session outage, deallocate (abend), or failure of the resource manager.	Work unit is ended.
ADAAERR	212	Resource no longer available due to failure of the adapter.	Work unit is ended.

ADAEWEOC (End of Command) Action

The SPM drives the resource adapter for this action because the application has ended normally. CMS end-of-command processing for CRR includes:

- Closing all open SFS files and directories
- Committing all work units
- Discarding all work units obtained by the DMSGETWU (Get Work Unit ID) CSL routine.

Table 52 on page 282 lists (in order of increasing severity) the response codes your resource adapter can return for an ADAEWEOC action call, and the resulting action taken by the SPM for each response.

Table 52. Resource Adapter Exit Routine Response Codes for the ADAEWEOC (End of Command) Sync Point Action Call

Response Code	Macro Value	Resource's Reason for Issuing This Response	Action Taken by the SPM
ADAOK	100	Default response.	Work unit is ended.
ADARF	8	Resource no longer available due to session outage, deallocate (abend), or failure of the resource manager.	Work unit is ended.
ADAAERR	212	Resource no longer available due to failure of the adapter.	Work unit is ended.

ADAEWABN (CMS Command Abend) Action

The SPM drives the resource adapter for this action because the application has ended abnormally. CMS action at CMS command abend for CRR includes:

- Calling the SPM to back out all active work units.
- Calling the SPM so that all existing work units can be cleaned up. The SPM calls end-of-work-unit exit processing with action value ADAEWABN for all existing work units (including work units where resynchronization has been initiated).

Table 53 on page 282 lists (in order of increasing severity) the response codes your resource adapter can return for an ADAEWABN action call, and the resulting action taken by the SPM for each response.

Table 53. Resource Adapter Exit Routine Response Codes for the ADAEWABN (CMS Command Abend) Sync Point Action Call

Response Code	Macro Value	Resource's Reason for Issuing This Response	Action Taken by the SPM
ADAOK	100	Default response.	Work unit is ended.

Table 53. Resource Adapter Exit Routine Response Codes for the ADAEWABN (CMS Command Abend) Sync Point Action Call (continued)

Response Code	Macro Value	Resource's Reason for Issuing This Response	Action Taken by the SPM
ADARF	8	Resource no longer available due to session outage, deallocate (abend), or failure of the resource manager.	Work unit is ended.
ADAAERR	212	Resource no longer available due to failure of the adapter.	Work unit is ended.

ADAEWSS (End of CMS Subset) Action

The SPM drives the resource adapter for this action because an application started in CMS subset mode has ended normally. CMS action at CMS end-of-subset for CRR includes:

- Closing all SFS files and directories opened during subset mode and still open.
- Committing all active subset mode work units.

CMS discards all subset mode work units at the end of subset mode.

Table 54 on page 283 lists (in order of increasing severity) the response codes your resource adapter can return for an ADAEWSS action call, and the resulting action taken by the SPM for each response.

Table 54. Resource Adapter Exit Routine Response Codes for the ADAEWSS (End of CMS Subset) Sync Point Action Call

Response Code	Macro Value	Resource's Reason for Issuing This Response	Action Taken by the SPM
ADAOK	100	Default response.	Work unit is ended.
ADARF	8	Resource no longer available due to session outage, deallocate (abend), or failure of the resource manager.	Work unit is ended.
ADAAERR	212	Resource no longer available due to failure of the adapter.	Work unit is ended.

ADABORQF (Backout-Required Function) Exit

The possible actions in the backout-required exit are:

- ADABRQBO (Backout)
- ADABRQRF (Resource Failure)
- ADABRQDA (Deallocate Abend).

The backout-required exit tells the resource adapter to ready itself and its resource manager for a subsequent backout or deallocate (abend) request from CRR. Some other resource has called the DMSSETR (Set Received) CSL routine to report an error outside of sync point processing. See “Backout Indications” on page 285. Note that the purpose of this exit is to put the resource in the proper state for a backout. It is not to *perform* a backout of the resource.

When a resource adapter is called for this function, it might want to prevent any further updates to its resource. This processing is architected for SNA LU 6.2 protected conversations. Other types of resources may not require any backout-required processing. The SPM will not drive a resource adapter for a backout-required exit if the SPM previously received an indication of resource failure, or deallocate abend, or backout for that resource since the last sync point.

Because the backout-required exit is called from the interrupt handler, processing within the exit is very restricted:

- No asynchronous activity is allowed.
- No APPC communication can take place.
- NO SVCs may be issued.
- The exit may not enable itself for interrupts.

Therefore, you can write a separate CSL routine to handle this exit.

ADABRQBO (Backout) Action

A backout indication was received on a verb outside of synchronization point processing. A participating resource's changes were backed out, but the resource is still available.

Table 55 on page 284 lists (in order of increasing severity) the response codes your resource adapter can return for an ADABRQBO action call, and the resulting action taken by the SPM for each response.

Table 55. Resource Adapter Exit Routine Response Codes for the ADABRQBO (Backout Required) Sync Point Action Call

Response Code	Macro Value	Resource's Reason for Issuing This Response	Action Taken by the SPM
ADAOK	100	Default response.	Work unit is put into a backout-required state.
ADARF	8	Resource no longer available due to session outage, deallocate abend, or failure of the resource manager.	Work unit is put into a backout-required state.
ADAAERR	212	Resource no longer available due to failure of the adapter.	Work unit is put into a backout-required state.

ADABRQRF (Resource Failure) Action

One of the participating resources was lost. This will most commonly occur when a link to the resource manager is dropped. Resource changes were either backed out, or will be backed out by the time the resource becomes available again.

Table 56 on page 284 lists (in order of increasing severity) the response codes your resource adapter can return for an ADABRQRF action call, and the resulting action taken by the SPM for each response.

Table 56. Resource Adapter Exit Routine Response Codes for the ADABRQRF (Resource Failure) Sync Point Action Call

Response Code	Macro Value	Resource's Reason for Issuing This Response	Action Taken by the SPM
ADAOK	100	Default response.	Work unit is put into a backout-required state.
ADARF	8	Resource no longer available due to session outage, deallocate abend, or failure of the resource manager.	Work unit is put into a backout-required state.
ADAAERR	212	Resource no longer available due to failure of the adapter.	Work unit is put into a backout-required state.

ADABRQDA (Deallocate Abend) Action

An SNA LU 6.2 protected conversation received a deallocate abend indication. Resource changes were backed out and the resource is unavailable. This action was begun by CMS protected conversation support processing, not a DMSSETR call.

Table 57 on page 285 lists (in order of increasing severity) the response codes your resource adapter can return for an ADABRQDA action call, and the resulting action taken by the SPM for each response.

Table 57. Resource Adapter Exit Routine Response Codes for the ADABRQDA (Deallocate Abend) Sync Point Action Call

Response Code	Macro Value	Resource's Reason for Issuing This Response	Action Taken by the SPM
ADAOK	100	Default response.	Work unit is put into a backout-required state.
ADARF	8	Resource no longer available due to session outage, deallocate abend, or failure of the resource manager.	Work unit is put into a backout-required state.
ADAAERR	212	Resource no longer available due to failure of the adapter.	Work unit is put into a backout-required state.

Backout Indications

A resource adapter could receive, on a non-sync point verb, an indication that a backout or failure occurred in one of its resources. Because the backout or failure happened outside of a sync point, the other resources registered on the work unit, the SPM, and the application are all unaware of the event and must be notified. There are two ways that the resource adapter can do this:

- Notify the SPM by calling the DMSSETR (Set Received) CSL routine. The SPM will in turn call backout-required exits to the resource adapters for the resources on the work unit that registered for the backout-required function. This is the preferred (architected) method. For information about the format and content of DMSSETR, see the *z/VM: CMS Callable Services Reference*.

  The resource adapter must also return a return code to the application to indicate that a backout is needed, and any attempt to do a coordinated commit will result in a backout.
- Notify the other resources on the work unit by issuing a CRR or product-specific backout verb. The resource adapter must also return a return code to the application to indicate that a backout has occurred.

Detailed Error Passback Support

This support provides a way for the resource manager to return product-specific error codes and information to the application. The general sync point error codes do not supply product-specific information.

When registering a resource with the DMSREG CSL routine, the resource adapter can define an error block to hold warning and error data. The SPM's register function then allocates a buffer and places the buffer's length into the first four bytes. The remaining content must be managed and documented by the resource adapter.

Note:

1. Defining and using an error block is optional. The error block buffer length is an optional parameter in DMSREG. If this parameter is not specified, the resource adapter will get a buffer that has no room to store its error data.

2. The size of the error block you should define depends on the type of error information your resource manager generates and how much you want to store. For example, CMS uses a 284-byte buffer for SFS and a 104-byte buffer for protected conversations.

Error blocks should contain the TPN (sometimes called the resource ID) for which the error occurred and perhaps an error identifier. The resource manager might want to provide a CSL routine to convert the error block to distinct values. For example, CMS provides the DMSWUERR routine to convert SFS error data.

The SPM keeps track of the actual error data length for each error block and resets the actual error data length for all protected resources at the start of each sync point. The SPM passes the error block and the actual error data length every time it drives the resource adapter's exit. An error block is considered empty if the actual error data length is four, meaning that the block contains only the length of the block.

The resource adapter's exit processing must manage the error block content and the actual error data length. For example, if a precoordination exit puts a warning code in the error block, and the coordination exit finds another warning, the first warning should not be overlaid. Rather, the second warning should be appended to the first by using and incrementing the actual error data parameter field.

When the resource adapter unregisters (DMSUNREG), the SPM deallocates the error block, if it is empty. If the error block contains error data, SPM does not deallocate the error block until the next sync point.

CRR provides the CSL routine DMSGETER to support detailed error passback. The resource adapter can use this routine to retrieve all error blocks generated by a resource manager, identified by component ID or adapter exit name or both, since the start of the last commit or backout for a work unit. The component ID and adapter exit name can be used as arguments to get specific error records. For information about the format and content of DMSGETER, see the *z/VM: CMS Callable Services Reference*.

Resource Manager Interface with the CRR Recovery Server

CRR resynchronization has two interfaces. One interface is with the CRR recovery server operator. Operation of the CRR recovery server is described in the *z/VM: CMS File Pool Planning, Administration, and Operation*. The other interface is between the CRR recovery server and the resource managers that are participating in CRR. You must support this interface to enable a resource manager to participate.

The two major functions that a resource manager must handle in cooperation with the CRR recovery server are resynchronization initialization and resynchronization recovery. These functions include exchanges of data between the resource manager and the CRR recovery server. A resource manager participating in CRR must be able to allocate and accept APPC conversations in which the synchronization level (SYNC_LEVEL) is set to CONFIRM, which allows either partner to request confirmation on the path. The resource manager must also be able to handle APPC general data stream (GDS) Exchange Log Names and Compare States data flows within the allocated conversations.

Resource Manager Resynchronization Facilities

There might be some problems that CRR cannot resolve, or situations where you cannot wait for CRR to resolve the problem. Therefore, the resource manager must supply facilities that the operator can use to manually resolve these problems. There are at least three tasks that the operator must do:

1. Display information about work that is in a prepared state, waiting to be either committed or backed out. The display should also indicated work that has been committed or backed out by previous operator intervention, but the history has not yet been removed from the log. (SFS provides the QUERY PREPARED command for this purpose.)
2. Force a commit or backout of the prepared CMS work unit. These are called heuristic actions. (SFS provides the FORCE PREPARED command for this purpose.)

   Note: The resource manager must remember heuristic actions until it is certain that resynchronization processing is complete.
3. Display information about CRR recovery servers with which the resource manager has previously exchanged log names. (SFS provides the QUERY LOGTABLE command for this purpose.)

4. Remove the history of the forced work from the log. (SFS provides the ERASE LUNAME command for this purpose.)

For information about the SFS commands to use as a guide in writing the facilities for your resource manager, see the description of SFS participation in CRR in the *z/VM: CMS File Pool Planning, Administration, and Operation*.

Exchanging Log Names

Exchanging log names ensures that the resource manager and the CRR recovery server have consistent log data about the units of work that might require resynchronization processing. Log names are exchanged during resynchronization initialization and resynchronization recovery. LU names, TPNs, and status information are also exchanged. The Exchange Log Names GDS variable is used to pass the information.

- **Exchanging log names in resynchronization initialization:**

  To initiate the exchange of log names and notify the CRR recovery server of its readiness to accept resynchronization communications, the resource manager must send an Exchange Log Names request to the CRR recovery server before the first sync point precoordination exit opportunity. The CRR recovery server sends an Exchange Log Names reply. Each partner saves the other partner's log name, LU name, and TPN to use in later validations.

- **Exchanging log names in resynchronization recovery:**

  To ensure consistent completion of a sync point by a registered resource, the CRR recovery server sends an Exchange Log Names request to the resource manager. The resource manager must send an Exchange Log Names reply. The partners validate the exchange by comparing the exchanged log names, LU names, and TPNs with the previously saved data. These comparisons determine whether resynchronization processing can continue.

When a resource manager is starting up processing after a failure or shutdown, it must do whatever is required to ensure that log name validation is complete.

Table 58 on page 287 shows the format of the Exchange Log Names variable used for resynchronization processing of z/VM resource managers participating in CRR. This format is patterned after the SNA architected resynchronization program (06F2), but includes some revised fields. The Exchange Log Names variable used for resynchronization processing of protected conversations in z/VM follows the SNA architecture.

Table 58. Exchange Log Names GDS Variable for z/VM Resource Managers

Field	Parameter
bytes 1 and 2	*length1*
bytes 3 and 4	*gds_identifier*
byte 5	*function_status_indicator*
byte 6	*log_status_flag*
byte 7	*length2*
bytes 8 to *n*	*local_fully_qualified_luname*
byte *n*+1	*length3*
bytes *n*+2 to *p*	*tpn*
byte *p*+1	*length4*
bytes *p*+2 to *r*	*log_name_1*
byte *r*+1	*length5*
bytes *r*+2 to *s*	*log_name_2*

As shown in Table 58 on page 287, the Exchange Log Names variable contains 12 parameters (some are optional):

length1
is the length (s) in binary of the variable, including this length parameter.

gds_identifier
is the GDS identifier for the Exchange Log Names variable, X'1211'.

function_status_indicator
indicates the status of the Exchange Log Names function.

Value
Meaning

X'02'
Request (set by the initiator of the exchange)

X'08'
Reply—abnormal completion (log status or log name mismatch)

X'09'
Reply—normal completion

log_status_flag
indicates the status of the sender's log.

Value
Meaning

X'00'
Cold—the sender's log has not been used in any previous communication with this partner. That is, the target's current TPN or LU name does not match the log or is not found.

X'01'
Warm—the sender's log was used in a previous communication with this partner. That is, the target's current TPN and LU name match the data saved in the log.

A warm log could contain information about units of work that were active when a previous failure or shutdown occurred, or a warm log could be empty (contain no *active* transaction data related to this partner).

length2
is the length in binary of the *local_fully_qualified_luname* parameter. Values 0–17 are valid.

local_fully_qualified_luname
is the sender's fully qualified LU name, which consists of the network node ID (*netid*) and the LU name.

A resource manager using CPI Communications can get this value and its length by calling the XCELFQ (Extract Local Fully Qualified LU Name) routine. For information about XCELFQ, see the *Common Programming Interface Communications Reference (https://publibfp.dhe.ibm.com/epubs/pdf/c2643999.pdf)*.

A resource manager using the APPC/VM assembler interface can get this value and its length from the connect complete extended data provided by the CMSIUCV CONNECT macro. For information about CMSIUCV CONNECT, see the *z/VM: CMS Macros and Functions Reference*.

A non-z/VM resource manager must use its own method to get this value.

length3
is the length in binary of the *tpn* parameter. Values 1–24 are valid.

tpn
is the sender's transaction program name (TPN).

If the sender is a z/VM resource manager, this is the standard identifier by which it identified itself to the system, using the *IDENT system service or the XCIDRM (Identify Resource Manager) CPI Communications routine.

If the resource adapter provided PIP data when it registered for CRR, the standard TPN should be used in this field. If the resource adapter provided a resource recovery TPN in its registration, the resource recovery TPN should be used in this field.

A non-z/VM resource manager must use its own method to get this value.

length4
is the length in binary of the *log_name_1* parameter. Values 1–64 are valid.

log_name_1
is the sender's log name.

length5
is the length in binary of the *log_name_2* parameter. Values 1–64 are valid. This parameter is required only if the following parameter is included.

log_name_2
is the saved log name for the partner. This parameter is included in the Exchange Log Names request only if the log status is warm. This parameter is not included in the Exchange Log Names reply.

Comparing States

When a sync point failure occurs, the CRR recovery server attempts to complete the action that was in progress on the CRR logical unit of work (also called a transaction), which is identified by an LUWID. The CRR recovery server and the resource manager must compare the sync point states of their respective logical units of work. The Compare States GDS variable is used to pass the information. The CRR recovery server sends the resource manager a Compare States request that specifies the LUWID and the desired sync point state (backout or committed). The resource manager must try to achieve that same state in its own logical unit of work. The resource manager then sends a Compare States reply to the CRR recovery server to indicate the sync point state in the resource and to report any heuristic actions.

Table 59 on page 289 shows the format of the Compare States variable used for resynchronization processing of z/VM resource managers participating in CRR. This format is patterned after the SNA architected resynchronization program (06F2), but includes some revised fields. The Compare States variable used for resynchronization processing of protected conversations in z/VM follows the SNA architecture.

Table 59. Compare States GDS Variable for z/VM Resource Managers

Field	Parameter
bytes 1 and 2	*length1*
bytes 3 and 4	*gds_identifier*
byte 5	*function_status_indicator*
bytes 6 and 7	*state_indicator*
byte 8	*length2*
bytes 9 to *p*	*luwid*
byte 9	*length3*
bytes 10 to *n*	*fully_qualified_luname*
bytes *n*+1 to *n*+6	*instance_number*
bytes *n*+7 to *p* (*n*+8)	*sequence_number*
byte *p*+1	*length4*
bytes *p*+2 to *r*	*recovery_token*
byte *r*+1	*length5*
bytes *r*+2 to *s*	*session_instance_id*

As shown in Table 59 on page 289, the Compare States variable contains 15 parameters (some are optional):

length1
is the length (s) in binary of the variable, including this length parameter.

gds_identifier
is the GDS identifier for the Compare States variable, X'1213'.

function_status_indicator
indicates the status of the Compare States function.

Value
Meaning

X'02'
Request (set by the CRR recovery server)

X'08'
Reply—abnormal completion (for example, a protocol violation)

X'09'
Reply—normal completion

state_indicator
indicates (in the request) the state of the CRR recovery server's logical unit of work, or (in the reply) the resource manager's response to that request.

Value
Meaning

X'0100'
Backout (Reset)

X'0300'
In-doubt (optimized last agent only)

X'0400'
Committed

X'0500'
Heuristic Backout (resource manager only)

X'0600'
Heuristic Committed (resource manager only)

X'0700'
Heuristic Mixed (CRR recovery server only)

The states that are recorded in the CRR recovery server's sync point log can affect and reflect these state indicators. See the explanations of the "Syncpoint State" and "Resync State" fields in the "Responses" section of the CRR QUERY LU operator command documentation in the *z/VM: CMS File Pool Planning, Administration, and Operation*.

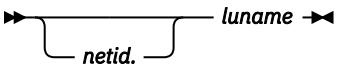
For an explanation of the states that are recorded in a resource manager's sync point log, see Table 63 on page 298.

length2
is the length in binary of the *luwid* field. Values 10–26 are valid.

luwid
identifies the logical unit of work. The LUWID consists of the following components:

length3
is the length in binary of the *fully_qualified_luname* parameter. Values 1–17 are valid.

fully_qualified_luname
is the SNA network node ID (optional) and LU name of the CRR recovery server that generated the LUWID. This parameter has the format



where *netid* has a length 0-8 (if a value is specified, it must be followed by a period as a delimiter) and *luname* has a length 1-8.

instance_number
is a unique 6-byte value that identifies the LUWID at the LU where it was generated.

sequence_number
is a 2-byte value that starts at 1 and is incremented by 1 following each commit or backout.

length4
is the length in binary of the *recovery_token* parameter. Values 0–8 are valid.

recovery_token
is a value that the resource manager may have assigned to identify this logical unit of work for this resource adapter. If defined, the value was passed to the SPM in the DMSREG routine when the resource adapter registered for sync point processing.

In a Compare States request, the CRR recovery server passes this value, if available, back to the resource manager to use in matching up processes during resynchronization recovery if the LUWID does not contain enough information to uniquely identify the process. The *recovery_token* and *length4* parameters are not required in the resource manager's Compare States reply.

length5
is the length in binary of the *session_instance_id* parameter. Values 0–8 are valid.

session_instance_id
is a value that the resource manager may have defined to identify the communication path between the resource manager and its resource adapter for this logical unit of work. The value is meaningful only if the resource adapter and resource manager are on different systems, communicating through AVS/VTAM. If defined, the value was passed to the SPM in the DMSREG routine when the resource adapter registered for sync point processing.

In a Compare States request, the CRR recovery server passes this value, if available, back to the resource manager so the resource manager can stop all activity on that communication path and deallocate. The resource manager may also use this value to identify the logical unit of work if the LUWID does not contain enough information to uniquely identify it. The *session_instance_id* and *length5* parameters are not required in the resource manager's Compare States reply.

How the Recovery Token and Session Instance ID Are Used

In a transaction with two partners, the partners by definition share the same LUWID value. If both partners access the same resource, then it is possible for both to have a task for that resource manager requiring resynchronization processing. If resynchronization recovery is necessary, the LUWID alone cannot distinguish between the tasks. In the Compare States request, the CRR recovery server supplies a recovery token or a session instance ID or both to help the resource manager identify the task.

The recovery token is a unique value that the resource manager can assign to each unit of work for each resource adapter. The resource manager passes the value, if defined, to the resource adapter, which passes it to the SPM in CRR registration (DMSREG). In resynchronization recovery, the CRR recovery server passes the recovery token, if available, back to the resource manager to use together with the LUWID as search arguments to find the matching task.

The session instance ID is a value that the resource manager can assign to identify the conversation between the resource manager and the resource adapter for the logical unit of work. A session instance ID is assigned only if the conversation is through AVS/VTAM. The resource manager passes the value, if defined, to the resource adapter, which passes it to the SPM in CRR registration (DMSREG). In resynchronization recovery, the CRR recovery server passes the session instance ID, if available, back to the resource manager to identify the task.

Resynchronization Initialization

The resource manager starts the resynchronization initialization function to:

- Exchange log names and other information with the CRR recovery server prior to participating in a sync point
- Notify the CRR recovery server of its presence and readiness to accept resynchronization communications (sometimes called a "shoulder tap").

The resource manager must complete resynchronization initialization before the first sync point is requested for a logical unit of work in which the resource manager is participating (specifically, before processing goes beyond the first precoordination exit call from the SPM).

Note: If your resource manager is participating in "limp mode" (that is, without a CRR recovery server), no exchange of log names is necessary or possible.

Resynchronization Initialization Data Flow

The following list summarizes the general sequence of events in resynchronization initialization. For examples of the communication flow, see Appendix I, “CRR Communications Examples,” on page 583.

1. The resource adapter allocates a conversation with the resource manager and passes the CRR recovery server's TPN and log name in a data buffer. (The resource adapter previously obtained these values by calling the CSL routine DMSGETRS. See “Getting Information about the CRR Recovery Server” on page 259.)
2. The resource manager must first determine if an exchange of log names with the CRR recovery server is required. The resource manager looks for:
   a. An entry for the CRR recovery server in the resource manager's log, using the following data as search arguments:
      - CRR recovery server's locally known LU name, obtained from the allocate data for the conversation that the resource adapter allocates with the resource manager
      - CRR recovery server's TPN and log name, passed to the resource manager by the resource adapter.

      Note:

      i) In VM, a resource manager could be accessed by the resource adapters on a particular processor through more than one LU. Therefore, the resource manager's log could have multiple entries containing the same CRR recovery server log name, but each with a different LU name.

      ii) If there is an entry for the locally known LU name and TPN, but the log name is different from the one passed by the resource adapter, the associated CRR recovery server has cold-logged (started up with a new log). The resource manager's log is considered warm, but the resource manager must initiate an Exchange Log Names request to give its own log name to the cold-logged CRR recovery server.

   b. A local indication that log names were exchanged. This indication can be a local flag associated with each log name record, a local caching of the log name record, or some other device. Whatever technique you use, its purpose is to determine if the resource manager has exchanged log names with the CRR recovery server during the resource manager's current activation.

      In cases where a resource manager could accidentally use the wrong log (as opposed to intentionally cold-logging), such as where logs are mounted or archived, it is important to force an exchange of log names with the CRR recovery server at least once for each activation of the resource manager to:

      - Catch a warm/warm log name mismatch, where it is possible that either the CRR recovery server's log or the resource manager's log is the wrong one.
      - Avoid having to check for this warm/warm mismatch on every allocation—just once each time the resource manager is activated.

- Accomplish an exchange of log names in case the CRR recovery server has erased log entries for the resource manager without cold-logging.

If conditions a and b are met and all the values match, the resource manager has previously been in communication with the CRR recovery server and no new exchange of log names is required.

If either condition is not met or any value does not match, the resource manager must initiate the exchange of log names, as described in the following steps. If the CRR recovery server's locally known LU name did not match the log or was not found, the resource manager must hold the value to be saved in the resource manager's log if the exchange of log names is successful.

3. The resource manager allocates an APPC conversation with the CRR recovery server, using CPI Communications (SAA communications) routines or APPC/VM assembler interface macros. The sync level must be set to CONFIRM.

   For information about CPI Communications (the SAA communications interface), see the *Common Programming Interface Communications Reference (https://publibfp.dhe.ibm.com/epubs/pdf/c2643999.pdf)*. For information about APPC/VM, see the *z/VM: CMS Macros and Functions Reference*.
4. The resource manager formulates and sends to the CRR recovery server an Exchange Log Names request that contains:
   - Log status, indicating whether the resource manager's log is cold or warm relative to the CRR recovery server.
     - If the CRR recovery server's locally known LU name (obtained from the allocate data) or TPN (obtained from the resource adapter) does not match the log or is not found, the resource manager's log status is cold.
     - If the locally known LU name and TPN match the log, the resource manager's log status is warm.
   - Resource manager's fully qualified LU name and TPN
   - Resource manager's log name
   - (Warm log only) CRR recovery server's log name saved from previous interactions.
5. The CRR recovery server receives the Exchange Log Names request and compares the resource manager's fully qualified LU name and TPN in the request with those saved in the recovery server's log to determine its own log status:
   - If the values match, the CRR recovery server's log status is warm.
   - If either value does not match, or is not found, the CRR recovery server's log status is cold.

   The CRR recovery server's subsequent actions are shown in Table 60 on page 293.

Table 60. Resynchronization Initialization Exchange Log Names Request. This table describes the CRR recovery server's actions after receiving the request from the resource manager.

Resource Manager's Log Status	Recovery Server's Log Status	Recovery Server's Actions
COLD or WARM	COLD	The recovery server saves the resource manager's fully qualified LU name, TPN, and log name in its log and sends an Exchange Log Names reply to the resource manager indicating normal completion of the request.

Table 60. Resynchronization Initialization Exchange Log Names Request. This table describes the CRR recovery server's actions after receiving the request from the resource manager. *(continued)*

Resource Manager's Log Status	Recovery Server's Log Status	Recovery Server's Actions
COLD	WARM	The recovery server checks the work unit records in its log that relate to this resource manager: • If the records do not contain active data, the recovery server updates the resource manager's log name saved in the log and sends an Exchange Log Names reply to the resource manager indicating normal completion of the request. • If the records contain active data, the recovery server issues CMS error message 3311E to the recovery server operator and sends an Exchange Log Names reply to the resource manager indicating abnormal completion of the request. The recovery server operator must manually resolve the error condition by forcing the active data from its log.
WARM	WARM	The recovery server compares the resource manager's log name sent in the request with the name that the recovery server has saved in its log. The recovery server also validates its own log name, if specified in the request. • If the log names match, the recovery server sends an Exchange Log Names reply to the resource manager indicating normal completion of the request. • If the log names do not match, the recovery server issues CMS message 3312E to the recovery server operator and sends an Exchange-Log-Names reply to the resource manager indicating abnormal completion of the request. The log name mismatch could be caused by one partner using the wrong log. If so, that partner must be restarted with the correct log. If the correct log cannot be supplied, one or both partners must be cold-started.

6. The CRR recovery server formulates and sends to the resource manager an Exchange Log Names reply that contains:
   - Function status, indicating normal or abnormal completion of the Exchange Log Names function
   - Log status, indicating whether the recovery server's log is cold or warm relative to the requesting resource manager
   - Recovery server's fully qualified LU name and TPN
   - Recovery server's log name.
7. The resource manager receives the Exchange Log Names reply. The resource manager's subsequent actions are shown in Table 61 on page 294.
8. The CRR recovery server deallocates the conversation.

Table 61. Resynchronization Initialization Exchange Log Names Reply. This table describes the resource manager's actions after receiving the reply from the CRR recovery server.

Recovery Server's Log Status	Reply Function Status	Resource Manager's Log Status	Resource Manager's Actions
COLD or WARM	NORMAL	COLD	The resource manager saves or updates the recovery server's locally known LU name, fully qualified LU name, TPN, and log name in its log and sends an explicit APPC confirmation to the recovery server.

Table 61. Resynchronization Initialization Exchange Log Names Reply. This table describes the resource manager's actions after receiving the reply from the CRR recovery server. *(continued)*

Recovery Server's Log Status	Reply Function Status	Resource Manager's Log Status	Resource Manager's Actions
COLD	NORMAL	WARM	The resource manager checks the work unit records in its log that relate to the recovery server: • If the records do not contain active data, the resource manager updates the recovery server's log name saved in the log and sends an explicit APPC confirmation to the recovery server. • If the records contain active data, the resource manager issues a product-specific equivalent to CMS message 3373E to the resource manager operator and does a deallocate (abend). The resource manager's participation in sync points must be delayed until the error condition is resolved. The resource manager operator should contact the recovery server operator to determine the reason for the status mismatch. The resource manager operator might have to manually force some units of work from its log. See “Resource Manager Resynchronization Facilities” on page 286.
WARM	NORMAL	WARM	The resource manager compares the recovery server's log name sent in the reply with the name that the resource manager has saved in its log: • If the log names match, the resource manager sends an explicit APPC confirmation to the recovery server. • If the log names do not match, the resource manager issues its equivalent to CMS message 3372E to the resource manager operator and does a deallocate (abend). The resource manager's participation in sync points must be delayed until the error condition is resolved. The resource manager operator should contact the recovery server operator to determine the reason for the log name mismatch. The resource manager or the recovery server might be using the wrong log, and should be restarted with the correct log. If the correct log cannot be supplied, both partners must be cold-started.

Table 61. Resynchronization Initialization Exchange Log Names Reply. This table describes the resource manager's actions after receiving the reply from the CRR recovery server. *(continued)*

Recovery Server's Log Status	Reply Function Status	Resource Manager's Log Status	Resource Manager's Actions
WARM	ABNORMAL	COLD or WARM	The resource manager issues its equivalent to CMS message 3371E to the resource manager operator and does a deallocate (abend). The resource manager's participation in sync points must be delayed until the error condition is resolved. The resource manager operator should contact the recovery server operator to determine the reason for the error. If the problem is a log name mismatch, one of the partners might be using an incorrect log and should be restarted with the correct log. If the correct log cannot be supplied, both partners must be cold-started.

Resynchronization Recovery

The CRR recovery server initiates the resynchronization recovery function to ensure consistent completion of the sync point by all registered resources for which data was logged. Using information stored in its log, the CRR recovery server determines which resource managers should be included in the recovery and allocates APPC conversations with them.

To allocate an APPC conversation with a resource manager, the CRR recovery server uses information that the resource manager provided when its resource adapter registered with the SPM (using the CSL routine DMSREG). For example, the resource manager (resource adapter) had to provide one of the following types of information during registration:

- Resource recovery TPN

  The resource manager can assign a special TPN for recovery. The CRR recovery server then uses this resource recovery TPN when allocating the resynchronization recovery conversation with the resource manager. This lets the resource manager know that the conversation is for resynchronization and not a normal data request. If a resource recovery TPN is assigned, the resource manager must use the *IDENT system service to make this ID known as a global resource. For more information about *IDENT, see *z/VM: CP Programming Services*.
- Program Initialization Parameters (PIP data)

  If a resource recovery TPN is not assigned, the resource manager must supply PIP data. When allocating a conversation with the resource manager for resynchronization, the CRR recovery server uses the resource manager's TPN but includes the PIP data to let the resource manager know that the conversation is for resynchronization. The definition and validation of the PIP data is the responsibility of the resource manager. For more information about PIP data, see *z/VM: CP Programming Services*.

The resynchronization recovery transaction between the CRR recovery server and the resource manager consists of two functions:

- Exchanging log names

  The CRR recovery server initiates this exchange with the resource manager to ensure that the data they saved from resynchronization initialization is still valid. See “Exchanging Log Names” on page 287.
- Comparing states

  The CRR recovery server initiates this exchange with the resource manager to compare the state of the CRR logical unit of work with the state of the resource manager's logical unit of work.

Resynchronization Recovery Data Flow

The following list summarizes the general sequence of events in resynchronization recovery. For examples of the communication flow, see Appendix I, “CRR Communications Examples,” on page 583.

1. The CRR recovery server allocates an APPC conversation with the resource manager. The sync level is set to CONFIRM.

   Note: If the resource adapter registered with a resource recovery TPN, the CRR recovery server establishes a conversation with that TPN. Otherwise, the CRR recovery server passes PIP data on the allocation so the resource manager will recognize that the incoming conversation is for resynchronization.
2. The CRR recovery server formulates the Exchange Log Names and Compare States requests and passes them in the same buffer to the resource manager:
   - The Exchange Log Names request contains:
     - Recovery server's fully qualified LU name and TPN
     - Recovery server's log status (warm)
     - Recovery server's log name
     - Expected (saved) log name for the resource manager.
   - The Compare States request contains:
     - State of the CRR logical unit of work (committed or backout). The CRR recovery server will attempt to complete the action that was in process when the failure occurred, as indicated in its log records.
     - LUWID
     - Recovery token or session instance ID or both, if specified in the resource registration (DMSREG). These values can be used to uniquely identify the logical unit of work being resynchronized. (For more information, see “How the Recovery Token and Session Instance ID Are Used” on page 291).
3. The resource manager receives the Exchange Log Names and Compare States requests. First, the resource manager processes the Exchange Log Names request, as shown in Table 62 on page 297.

Table 62. Resynchronization Recovery Exchange Log Names Request. This table describes the resource manager's actions after receiving the request from the CRR recovery server.

Recovery Server's Log Status	Resource Manager's Log Status	Resource Manager's Actions
WARM	COLD	The resource manager holds the recovery server's log name from the request but does not update its own log or process the Compare States request. The resource manager sends an Exchange Log Names reply to the recovery server indicating cold log status and normal completion of the request. The resource manager waits for indication of a deallocate (abend) by the recovery server, then does a deallocate (normal).

Table 62. Resynchronization Recovery Exchange Log Names Request. This table describes the resource manager's actions after receiving the request from the CRR recovery server. *(continued)*

Recovery Server's Log Status	Resource Manager's Log Status	Resource Manager's Actions
WARM	WARM	The resource manager compares the recovery server's log name in the request with the name that the resource manager has saved in its log. The resource manager also validates its own log name specified in the request. • If the log names match, the resource manager formulates (but does not send) an Exchange Log Names reply indicating normal completion of the request. The resource manager deallocates the conversation with the resource adapter used for the logical unit of work identified in the Compare States request (if that conversation is still allocated), then processes the Compare States request (Step "4" on page 298). After the resource manager completes the Compare States processing, it sends both replies in the same buffer. • If the log names do not match, the resource manager issues its equivalent to CMS message 3372E to the resource manager operator and sends an Exchange Log Names reply to the recovery server indicating abnormal completion of the request. The resource manager does not process the Compare States request. The resource manager waits for indication of a deallocate (abend) by the recovery server, then does a deallocate (normal). The resource manager operator should contact the recovery server operator to determine the reason for the log name mismatch. The resource manager might be using the wrong log. If so, the resource manager should be restarted with the correct log. If the correct log cannot be supplied, the resource manager must be cold-started. If the recovery server is using the wrong log and cannot locate the correct log, the resource manager might have to manually force some units of work from its log. See "Resource Manager Resynchronization Facilities" on page 286.

4. If the log name exchange was satisfactory, the resource manager processes the Compare States request, as shown in Table 63 on page 298. Some resource managers might not keep enough information to distinguish all the logical-unit-of-work states shown. The resource manager can handle these logical-unit-of-work states as it pleases, as long as the resource is restored to a consistent state.

   In addition, a resource manager must remember heuristic actions associated with a unit of work until it is certain that resynchronization processing is complete. Normally, the CRR recovery server will tell the resource manager what information can be discarded. However, if the CRR recovery server cold-logs, the resource manager will not receive any such notification. In that case, the resource manager must use its own facilities to determine what information should be discarded from its log. See "Resource Manager Resynchronization Facilities" on page 286.

Table 63. Resynchronization Recovery Compare States Actions

LUWID State at Resource Manager	LUWID State Sent by Recovery Server	
	Backout	Committed
	Resource Manager's Actions	
LUWID Not Found	Send normal completion reply indicating Backout state.	Send normal completion reply indicating Backout state.
Syncpoint Pending	Send normal completion reply indicating Backout state.	Send normal completion reply indicating Backout state.

Table 63. Resynchronization Recovery Compare States Actions (continued)		
LUWID State at Resource Manager	**LUWID State Sent by Recovery Server**	
	Backout	**Committed**
	Resource Manager's Actions	
Backout (Reset)	Send normal completion reply indicating Backout state.	Protocol violation—send abnormal completion reply.
In-doubt (Prepared)	Drive backout of resource and send normal completion reply indicating Backout state.	Drive commit of resource and send normal completion reply indicating Committed state.
Committed	Protocol violation—send abnormal completion reply.	Send normal completion reply indicating Committed state.
Heuristic Backout	Send normal completion reply indicating Heuristic Backout state.	Send normal completion reply indicating Heuristic Backout state. Operator notification is appropriate.
Heuristic Committed	Send normal completion reply indicating Heuristic Committed state. Operator notification is appropriate.	Send normal completion reply indicating Heuristic Committed state.

Note: The SFS resource manager writes to the sync point log *after* completing the sync point actions, thereby avoiding certain intermediate states such as Syncpoint Pending and Committed. Also, the sequence of sync point operation and logging is such that SFS resynchronization actions can safely mirror the LUWID states sent by the CRR recovery server (in resynchronization recovery) when there is an LUWID Not Found state for the SFS log entry. This would not be possible if SFS wrote to the log *before* processing the sync point actions. The Compare States actions are therefore slightly different for SFS, as shown in Table 64 on page 299.

Table 64. SFS Resource Manager's Compare States Actions		
LUWID State at SFS Resource Manager	**LUWID State Sent by Recovery Server**	
	Backout	**Committed**
	SFS Resource Manager's Actions	
LUWID Not Found	Send normal completion reply indicating Backout state.	Send normal completion reply indicating Committed state.
In-doubt (Prepared)	Drive backout of resource and send normal completion reply indicating Backout state.	Drive commit of resource and send normal completion reply indicating Committed state.
Heuristic Backout	Send normal completion reply indicating Heuristic Backout state.	Send normal completion reply indicating Heuristic Backout state. Operator notification is appropriate.
Heuristic Committed	Send normal completion reply indicating Heuristic Committed state. Operator notification is appropriate.	Send normal completion reply indicating Heuristic Committed state.

5. The resource manager formulates the Compare States reply, then sends the Exchange Log Names and Compare States replies to the CRR recovery server in the same buffer:
   - The Exchange Log Names reply contains:
     - Function status (normal or abnormal completion)
     - Log status (warm or cold)

  - Resource manager's fully qualified LU name and TPN

    Note: If the resource adapter registered with a resource recovery TPN, the resource manager must use that TPN here.
  - Resource manager's log name.
- The Compare States reply contains:
  - Function status (normal or abnormal completion)
  - State that the resource manager was able to achieve in its logical unit of work, including heuristic actions.

6. The CRR recovery server processes the Exchange Log Names reply as shown in Table 65 on page 300.

Table 65. Resynchronization Recovery Exchange Log Names Reply. This table describes the CRR recovery server's actions after receiving the reply from the resource manager.

Resource Manager's Log Status	Reply Function Status	Recovery Server's Log Status	Recovery Server's Actions
COLD	NORMAL	WARM	The recovery server issues CMS message 3311E to the recovery server operator and does a deallocate (abend). The recovery server operator must resolve the active records in its log.
WARM	NORMAL	WARM	The recovery server compares the resource manager's log name sent in the reply with the name that the recovery server has saved in its log. • If the log names match, the recovery server confirms completion of the resync transaction (including the Compare States flow). • If the log names do not match, the recovery server issues CMS message 3312E to the recovery server operator and does a deallocate (abend). The recovery server then goes into a timed-wait state. In this state, the recovery server waits a specific interval (specified by the RESYNCINTERVAL start-up parameter in the DMSPARMS file), then retries the resync transaction. The recovery server will keep cycling in this manner until the situation either resolves itself or the recovery server operator intervenes to resolve it manually. The log name mismatch could be caused by one partner using the wrong log. If so, that partner must be restarted with the correct log. If the correct log cannot be supplied, the resource manager must be cold-started.

Table 65. Resynchronization Recovery Exchange Log Names Reply. This table describes the CRR recovery server's actions after receiving the reply from the resource manager. *(continued)*

Resource Manager's Log Status	Reply Function Status	Recovery Server's Log Status	Recovery Server's Actions
WARM	ABNORMAL	WARM	The recovery server issues CMS message 3310E to the recovery server operator and does a deallocate (abend). The recovery server then goes into a timed-wait state. In this state, the recovery server waits a specific interval (specified by the RESYNCINTERVAL start-up parameter in the DMSPARMS file), then retries the resync transaction. The recovery server will keep cycling in this manner until the situation either resolves itself or the recovery server operator intervenes to resolve it manually. The log name mismatch could be caused by one partner using the wrong log. If so, that partner must be restarted with the correct log. If the correct log cannot be supplied, one partner must be cold-started. The other partner might have to manually force some units of work from its log.

Note: If the CRR recovery server discovers a protocol violation in the Exchange Log Names or Compare States reply, the CRR recovery server issues CMS message 3313E, does a deallocate (abend), and goes into timed-wait state. The resource manager must correct the error and resend the replies.

7. If the CRR recovery server confirms completion of the resynchronization transaction, the resource manager resolves the records in its log relating to the transaction and confirms.
8. The CRR recovery server deallocates the conversation.

Forward Recovery

When writing the changes for your resource manager, you should consider including "forward recovery" capability, if the resource manager does not already provide it. If a resource manager with this capability has a media failure, it can recover all the data, including transactions that have been processed since the last backup. A resource manager without this capability can recover only up to the last backup. If one or more resource managers without this capability participate in CRR, a resource media failure might result in inconsistent data.

Using Protected Conversations

In designing and developing your resource manager's modifications to support CRR, there are certain cases where you can use or must use protected conversations. Two such cases are:

- Your resource manager is not itself directly maintaining the resources.

  Figure 45 on page 302 shows an example of a resource manager (RM1) that supports objects made up of files that are in SFS file pools maintained by other resource managers (RM2 and RM3). RM1 uses a protected conversation (PC) to communicate with the application. RM1 uses non-protected conversations (non-PC) to communicate with the other resource managers, and they use nonprotected conversations to communicate with their respective file pools.

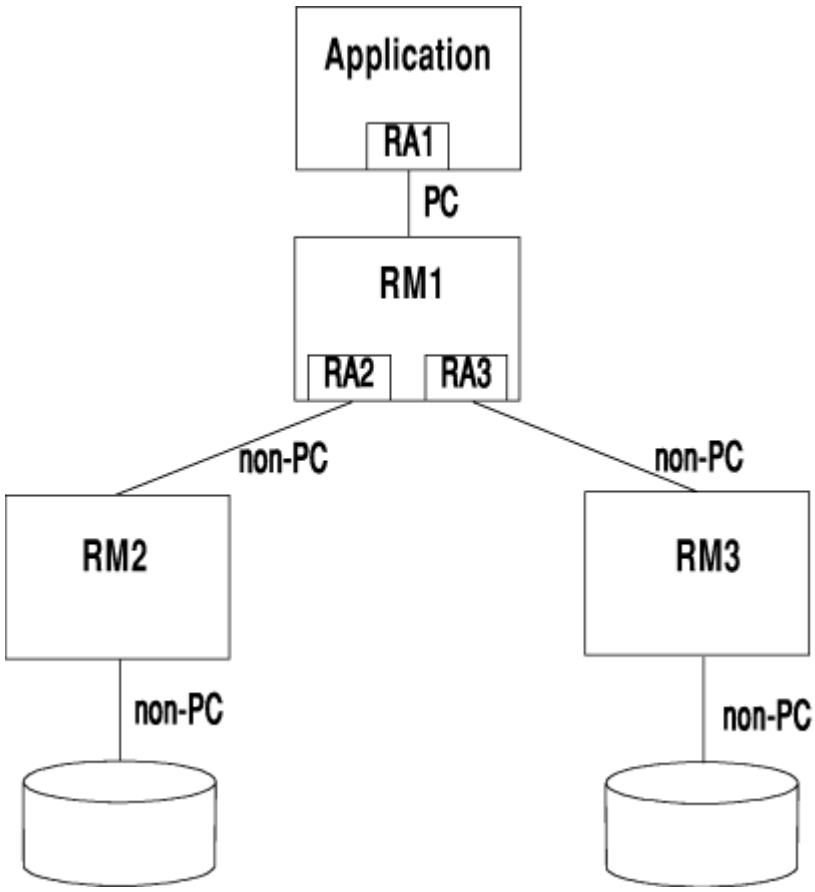


Figure 45. How a Resource Manager Not Directly Maintaining the Resources Uses Protected Conversations

- Your resource manager directly maintains the resources but is distributed (made up of two or more separate resource managers).

  Figure 46 on page 303 shows an example of a resource manager (RM1) that is the primary resource manager in a tree of resource managers. A single logical unit of work identifier (LUWID) must be maintained within all parts of the application and within all of the participating resource managers. In this case, protected conversations (PC) are required because they support the LUWID maintenance. RM1 uses protected conversations to communicate with the application and with the other resource managers in the tree. Each resource manager uses nonprotected conversations (non-PC) to communicate with its respective resource.

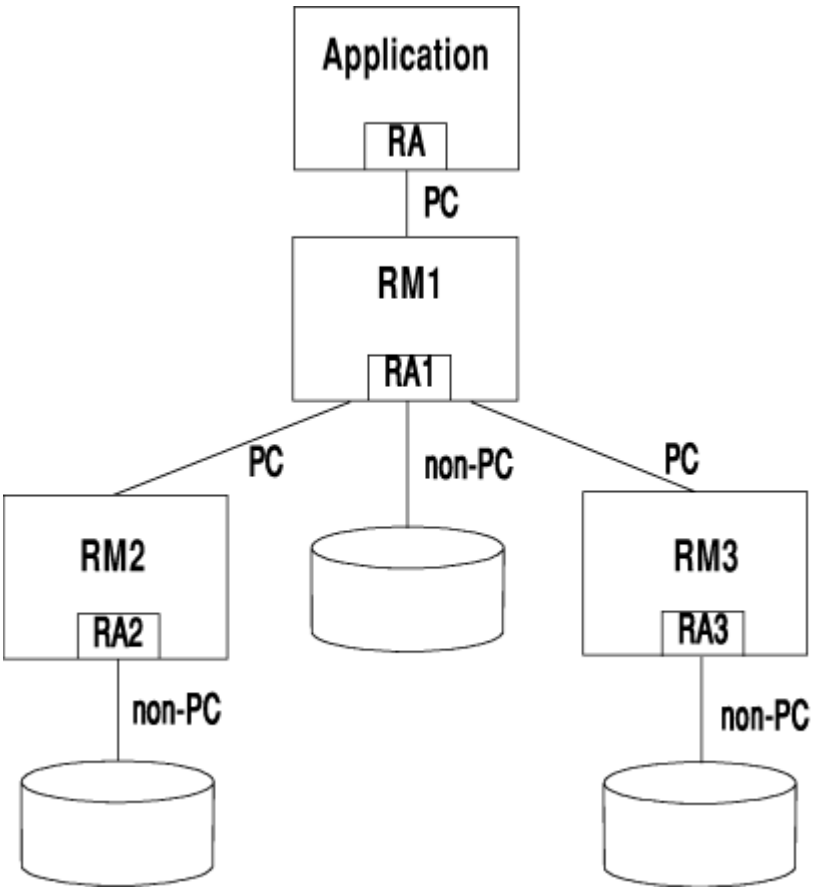


Figure 46. How Distributed Resource Managers Use Protected Conversations

When using protected conversations, there are some special rules for CRR participation:

- A resource adapter using protected conversations **must not register for the coordination exit**. Coordination for protected conversations is handled by CMS APPC support.
- The resource adapter should register for the precoordination exit. Within that exit, all data that the resource adapter controls that is related to the commit should be flushed to the resource manager.
- You should set the simple-commit flag OFF to ensure that CRR does a two-phase commit.
- During coordination, if your resource manager receives return codes on the protected conversation to commit or back out, it should issue the system commit or backout to call the SPM.
- The SPM handles the system commit or backout in a synchronous manner. That is, it does not return to your resource manager until the process is complete. If your resource manager supports multiple users, and you want to overlap the processes for your users, you should replace the CMS-supplied wait routine, DMSCWAIT, with your own CSL routine. This allows your resource manager to get control when the system is waiting for processing to complete within the CRR exits. For more information, see Chapter 17, “Writing a CRR Wait Routine for Multiuser Server Applications,” on page 251.
- If your resource manager does not directly maintain the resources (RM1 in Figure 45 on page 302), it is not required to register for CRR at all. However, the resource managers that do directly maintain the data (RM2 and RM3) must have resource adapters in your resource manager's virtual machine that register for CRR, including the coordination exit.

  When your resource manager receives a commit or backout return code on the protected conversation and issues the system commit or backout to the local SPM, the SPM communicates with all protected conversations and registered resource adapters to make sure that the work is committed or backed out in a coordinated way. The SPM does this without further participation by your resource manager.

  After the commit or backout is completed, the SPM returns to your resource manager. Your resource manager is now ready to receive or send the next message, end the conversation, and so on, depending on the protocol you have established between your resource manager and its resource adapter.

- If your resource manager directly maintains a resource in a distributed environment (see Figure 46 on page 303), it must have a resource adapter that registers for CRR, including the coordination exit, in the resource manager's own virtual machine. As the primary resource manager (RM1), it must also have an adapter that registers for CRR in the application's virtual machine.

  In this case, your resource manager prepares for the commit, actually does the commit or backout, and supports resynchronization. Your resource manager must support commit or backout of changes for an LUWID asynchronously after a termination. This resynchronization support is provided for any LUWID for which your resource manager was prepared to commit, and for which the system or subsystem terminated before the commit or backout could be completed.
- If the resource adapter registers for postcoordination, that function should be used only for tasks such as freeing buffers. Do not attempt to send any data.

Chapter 19. Creating and Manipulating the CMS Libraries

This chapter describes:

- How to create and manipulate macro libraries, text libraries, and load libraries
- How to use callable services libraries
- How to use ISPF/PDF libraries.

Most operating systems provide library facilities. These help you develop programs and maintain an orderly environment for managing your files. All the CMS library types have a similar structure. Each one contains one or more members and has an ***internal directory***. The library facilities use this directory to locate members. Because libraries are unlike other CMS files, you create, update, and use them differently than you do other CMS files.

The CMS libraries are:

Macro Library (MACLIB)
Macro libraries have a file type of MACLIB. They contain COPY files usually written in a high-level programming language or MACRO files usually written in assembler language. These files are referenced when you invoke either a compiler or assembler to process an application. Some MACLIBs are provided with the individual programming language compilers. These MACLIBs contain routines used during the compilation process. Therefore, before compiling or assembling your application, you may need to make these routines available to the compiler or assembler by issuing the GLOBAL command. See "Identifying Libraries to Be Searched" on page 45 for information on using the GLOBAL command to access MACLIBs. You can also use the MACLIB command to create or change the contents of a macro library.

Text Library (TXTLIB)
Text libraries have a file type of TXTLIB. They contain files (sometimes called object files) that are compiled or assembled. Some TXTLIBs are provided with the individual programming language product you are using. Therefore, before executing your application, you may need to make these TXTLIBs available to CMS by issuing the GLOBAL command. See "Resolving External References by Identifying Libraries" on page 52 for information on using the GLOBAL command to access TXTLIBs. You can also create your own TXTLIBs with routines written for use in one or more applications.

Load Library (LOADLIB)
Load libraries have a file type of LOADLIB. They contain executable load modules that have been compiled or assembled and link-edited.

Callable Services Library (CSL)
Callable services libraries have a file type of CSLLIB, CSLSEG, or TEXT, depending on whether the library is located on DASD, in a logical saved segment, or in the CMS nucleus. Files within these libraries contain routines that are written in the assembler language. Your high-level language or assembler application can call these routines to perform a specific function. z/VM provides you with libraries named VMLIB and VMMTLIB that contain many routines your applications can call.

Interactive System Productivity Facility/Program Development Facility (ISPF/PDF)
An ISPF/PDF library can be a set of CMS files, MACLIBs, or TXTLIBs. Organizing this information in an ISPF/PDF library allows many people to share the code and data.

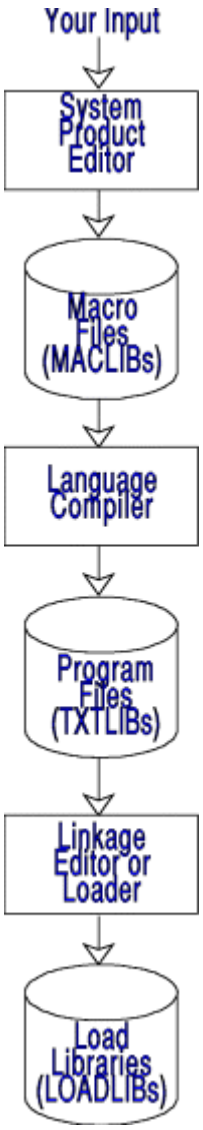


Figure 47. CMS Libraries

Creating and Manipulating Macro Libraries

A CMS macro library (MACLIB file type) contains one or more copy files (COPY file type) usually written in a high-level language or one or more macro files (MACRO file type) usually written in assembler language. A COPY file contains predefined source statements that are included in a source program when the COPY statement is encountered. When used in an assembler language program, macrodefinitions in a MACRO file generate code in line by referencing the macro name. These files are referenced when you invoke either one of the compilers or the assembler to process a program.

Use the MACLIB command to change or create the contents of MACLIBs. Use the GLOBAL command to identify the macro libraries to be searched for macros and copy files when processing source code.

A MACLIB is similar to an OS partitioned data set (PDS). It has individual members that you can create using the editor or that you can copy from other source files. For a member to be added to a MACLIB, it must be in a CMS file with a file type of COPY or MACRO. Use the file type COPY for files containing source code to be included in a MACLIB. The file type MACRO is usually used for assembler language macros.

CMS provides you with macro libraries on the system disk. These macro libraries contain various CMS and OS/MVS assembler language macros that you may want to use in your programs. However, you can use the MACLIB command, commands from the MACLIST screen, and some CMS commands to:

- Create a MACLIB

- List the contents of a MACLIB
- Add members to a MACLIB
- Replace members in a MACLIB
- Delete members from a MACLIB
- Compress a MACLIB
- Edit members of a MACLIB
- Print and display members of a MACLIB
- Manipulate members of a MACLIB.

Note: You should not use the OS/MVS STOW macro to create or modify MACLIBs; only use the MACLIB command to manipulate MACLIBs. You should not use any OS macros to create or update MACLIBs. OS Partitioned Data Sets and CMS MACLIBs interpret certain common fields differently. For this reason, OS macros, which were written to support OS conventions, do not create CMS libraries that can be used reliably.

A macro library file may reside in an SFS directory. If you perform MACLIB command functions on a macro library that resides in an SFS directory and the resulting library has no members, an empty macro library is maintained in the SFS directory to preserve any authorities to the macro library file. You should use only the MACLIB command to perform functions on an empty macro library.

Using System MACLIBs

The macro libraries on the system disk, supplied as part of the system, are:

- DMSGPI contains CMS macros that are programming interfaces. In prior releases, these macros were in DMSSP MACLIB and CMSLIB MACLIB, which no longer exist.
- DMSOM contains programming interface macros and 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.
- OSMACRO contains the macros that CMS provides for execution of programs using MVS interfaces.
- MVSXA contains the simulated MVS/XA versions of the OS/MVS macros for the execution of programs using MVS interfaces.
- OSMACRO1 contains the non-simulated versions of OS/MVS macros that are used only for assembly on CMS. Because CMS does **not** simulate these macros you should not use them in programs you intend to run on CMS.
- OSVSAM contains the subset of supported OS/VSAM macros.
- HCPGPI contains CP programming interface macros. The IUCV macro and the APPCVM macro are examples of macros that reside in this macro library.
- HCPPSI contains additional programming interface macros.

Creating a MACLIB

The GEN function of the MACLIB command generates a CMS macro library from input files specified on the command line. The input files must have a file type of either MACRO or COPY.

For example, to create the macro library, TESTMAC MACLIB A1, from the following files:

TEST1 MACRO A1
TEST2 MACRO A1
TEST3 MACRO A1
TEST4 COPY A1

enter:

```
maclib gen testmac test1 test2 test3 test4
```

If a library named TESTMAC MACLIB A1 already exists, it is replaced by this new library.

When the input file is a COPY file, the member name(s) are taken from the name of the COPY file or from the *COPY statement(s) in the COPY file. When the input file is a MACRO file, the member name(s) are taken from macro prototype statements in the MACRO file. If a file contains more than one macro, the MACLIB command gets the library member names from the macro prototype statements of each macro in the file.

For example, suppose that several macrodefinitions, including one for TEST3 MACRO, are in the TEST1 MACRO file.

For example, the TEST1 MACRO file might look like this:

```
TEST1    MACRO
         ⋮
         MEND
TEST1A   MACRO
         ⋮
         MEND
TEST3    MACRO
         ⋮
         MEND
```

If you issue the same MACLIB GEN command shown earlier, TESTMAC MACLIB has the following members in this order:

TEST1
from TEST1 MACRO A1

TEST1A
from TEST1 MACRO A1

TEST3
from TEST1 MACRO A1

TEST2
from TEST2 MACRO A1

TEST3
from TEST3 MACRO A1

TEST4
from TEST4 COPY A1

The TEST3 macro, which appears in both the TEST1 MACRO file and the TEST3 MACRO file, now exists as two members in TESTMAC MACLIB. However, there is only one entry in the MACLIB directory. The MACLIB command does not check for duplicate macro names. Later, when a program requests TEST3 macro from TESTMAC MACLIB, it uses the first TEST3 macro it meets (from the TEST1 MACRO file).

Examining Contents of a MACLIB

To examine the contents of a macro library, you may use either the MACLIB or MACLIST command.

Using MACLIB Command

The MAP function of the MACLIB command lists information about members in a macro library. This information includes:

- Member name
- Size of the member
- Sequential position in the library.

You can obtain this information as a:

- File on your A-disk (the DISK option, the default)
- Spooled printer file (the PRINT option)
- Display on your terminal (the TERM option).

For example, to list the members of TESTMAC MACLIB, enter:

```
maclib map testmac
```

You can also retrieve information for specific members of a library by indicating the member names following the MAP operand. For example, to list the TEST1 members of TESTMAC MACLIB, enter:

```
maclib map testmac test1
```

The DISK option creates a file with the file type MAP. The file name is the same as the MACLIB being mapped. All three options erase any existing MAP file for the specified MACLIB.

If you want to place that information in the program stack, use the STACK option of the MAP operand. The information can be stacked FIFO (first-in, first-out) or LIFO (last-in, last-out). The default order when STACK is specified alone is FIFO. The options STACK, STACK FIFO, and FIFO are equivalent. The options STACK LIFO and LIFO are equivalent. For example:

```
maclib map testmac test2 test3 (stack fifo
```

stacks, in the program stack, the MAP output for the TEST2 and TEST3 members of TESTMAC in first-in, first-out order.

Using MACLIST Command

The MACLIST command displays a list of all members in the specified macro library. MACLIST provides you with an easy way to select and edit CMS MACLIB members. If you issue the MACLIST command on a macro library that resides in an SFS directory and the specified macro library does not contain any members, you will receive the following message:

```
DMSWML213W Library fn ft has no members
```

You can type commands that operate on member names in the list directly on the lines of the MACLIST display. When you press ENTER, all commands typed on the lines in the file displayed on the current screen are executed. Symbols can represent operands in the command to be executed. Symbols are needed if the command to be executed has operands or options that follow the file ID.

In the MACLIST environment, information that is normally provided by the MACLIB command (with the MAP option) is displayed under the control of XEDIT. You can use XEDIT subcommands to manipulate the list itself.

To create the MACLIB screen shown in Figure 48 on page 310, enter:

```
maclist testmac
```

Note that the members are sorted alphabetically by member name. Members with the same name are then sorted by index number (least to greatest).

```
 TESTMAC  MACLIST  A0  V 130  Trunc=130 Size=18 Line=1 Col=1 Alt=0
Cmd   Member name    Index     Records   Library name  Library type  Mode
      TEST1            190           6   TESTMAC       MACLIB        A1
      TEST1A           240          25   TESTMAC       MACLIB        A1
      TEST3            613          57   TESTMAC       MACLIB        A1
      TEST2            197          25   TESTMAC       MACLIB        A1
      TEST3            615          25   TESTMAC       MACLIB        A1
      TEST4            546          55   TESTMAC       MACLIB        A1









1= Help      2= Refresh  3= Quit    4= Sort(name) 5= Sort(index)  6= Sort(size)
7= Backward  8= Forward  9= FL /n  10=           11= XEDIT       12= Cursor
====>
                                                          X E D I T  1 File
```

Figure 48. Sample MACLIST Screen

Adding MACLIB Members

To add members to a macro library, you may use either the MACLIB or XEDIT command.

Using MACLIB Command

The ADD function of the MACLIB command adds members to a macro library. No checking is done for duplicate names, entry points, or CSECTs. The new member is added at the end of the library.

Suppose you want to add TEST5 COPY to the TESTMAC MACLIB. The command for this action looks like this:

```
maclib add testmac test5
```

TESTMAC MACLIB now contains the following members:

TEST1
from TEST1 MACRO A1

TEST1A
from TEST1 MACRO A1

TEST3
from TEST1 MACRO A1

TEST2
from TEST2 MACRO A1

TEST3
from TEST3 MACRO A1

TEST4
from TEST4 COPY A1

TEST5
from TEST5 COPY A1

If you perform the ADD function on an empty library, the GEN function will be performed on the macro library.

Using XEDIT Command

You could also add members to a macro library using the MEMBER option of the XEDIT command for a member that does not exist in the library. A new file is created with the file ID of *membername* MEMBER *fm*.

For example, you can add a new member TEST5 to the TESTMAC MACLIB by entering:

```
xedit testmac maclib (member test5
```

When you issue the FILE or SAVE command for this new member, the TESTMAC MACLIB directory is updated. The new member and the updated library directory are added to the end of the library. If the directory already contains a member with the same name as the one being saved, the old entry is blanked out, so that the updated member replaces the old version.

Replacing MACLIB Members

To replace members of a macro library, you may use either the MACLIB or XEDIT command.

Using MACLIB Command

The REP function replaces members in a macro library by deleting the directory entry for the macrodefinition in the specified library. It adds new macrodefinitions to the library and creates new directory entries.

Suppose you want to replace the TEST2 macro with a later debugged version or one with new features or code. The command line:

```
maclib rep testmac test2
```

causes the following actions:

1. The latest version of the TEST2 macro (in the file TEST2 MACRO A1) is added to the library.
2. The old directory entry for the last version of TEST2 is deleted from the library.
3. A new directory entry is created.

The physical order of members in the library is arranged so that the new version of TEST2 appears after the old version. The logical order (the one in which requests for macros are satisfied) is determined by the library directory entry—not by the physical position of the member in the library. The REP function causes the directory entry rather than the source code to be replaced.

Using XEDIT Command

You could also replace members of a macro library using the MEMBER option of the XEDIT command for an existing member of a library. The member is read into a file called *membername* MEMBER *fm* for you to edit.

For example, you can replace the member TEST2 of the TESTMAC MACLIB by entering:

```
xedit testmac maclib (member test2
```

When you issue the FILE or SAVE command for this changed member, the TESTMAC MACLIB directory is updated. The changed member and the updated library directory are added to the end of the library. If the directory already contains a member with the same name as the one being saved, the old entry is blanked out, so that the updated member replaces the old version.

Deleting MACLIB Members

To delete members of a macro library, you may use either the MACLIB or MACLIST command.

Using MACLIB Command

The DEL function deletes members from a macro library. What it does is remove the member name from the library directory so there are no unused entries. The macrodefinitions or copy code still takes up space in the library but cannot be accessed because it has been deleted in the library directory entry.

If a library contains two members with the same name, only the first member is deleted from the directory. Deleting the last remaining member of a MACLIB erases the entire MACLIB. However, if you perform the DEL function on a macro library that resides in a SFS directory and the resulting library has no members, an empty macro library is maintained in the SFS directory. This is done to preserve any authorities to the macro library file. You should use only the MACLIB command to perform functions on an empty macro library.

To delete the first version of the TEST3 macro, the one from the TEST1 file, enter:

```
maclib del testmac test3
```

The result is this:

TEST1
from TEST1 MACRO A1

TEST1A
from TEST1 MACRO A1

(TEST3
from TEST1: present but unavailable)

(TEST2
from TEST2: present but replaced)

TEST3
from TEST3 MACRO A1

TEST4
from TEST4 COPY A1

TEST2
from TEST2 MACRO A1, later version

If you have MACRO and COPY files (on any accessed minidisk or in a directory) with the same file name, the MACRO version is used when you invoke the MACLIB command.

Using MACLIST Command

The DISCARD command from the MACLIST screen deletes a member from a library. DISCARD is equivalent to the CMS command MACLIB DEL. DISCARD can either be typed in the command area of the line that describes the member you want discarded, or it can be entered from the command line (at the bottom of the screen).

Compressing a MACLIB

When you use the ADD, DEL, and REP functions repeatedly, the library ends up with *dead entries* or *nonmembers*. These are macros and copy code that remain in the library but are no longer used because they have no library directory entries. The COMP function compresses a library by deleting any macros or copy blocks that do not have library directory entries.

The MACLIB command does this by copying each member of the file to a new file, using the directory. The new file now has the temporary name of MACLIB CMSUT1. This name is always used, regardless of the original macro library file name. After all valid library members are copied to MACLIB CMSUT1, the old library is erased and the temporary CMSUT1 file is renamed with the old library name.

To continue our example, the earlier results show that TESTMAC MACLIB now contains two nonmembers. One is the TEST2 macro that was replaced by a later version. The other is the TEST3 macro that was deleted. To save DASD space, you may want to compress TESTMAC to eliminate the two nonmembers by issuing the command:

```
maclib comp testmac
```

The resulting library contains the same valid members as those listed earlier. However, the ones in parentheses (the first version of TEST3 and the earlier version of TEST2) no longer occupy space in the MACLIB. Thus, the new TESTMAC MACLIB is smaller than the old one. It lost the two files plus two delimiter records. The directory size remains the same, because it was already compressed. The result is this:

TEST1
from TEST1 MACRO A1

TEST1A
from TEST1 MACRO A1

TEST3
from TEST3 MACRO A1

TEST4
from TEST4 COPY A1

TEST2
from TEST2 MACRO A1

If you perform the compress function on a macro library that resides in a SFS directory and the resulting library has no members, an empty macro library is maintained in the SFS directory. This is done to preserve any authorities to the macro library file.

Editing MACLIB Members

To edit members of a macro library, you may use either the MACLIST or XEDIT command.

Using MACLIST Command

The MACLIST command allows you to select and edit a CMS maclib member from the list. To edit a member, position the cursor on the line that contains the member to be edited and press PF11.

Using XEDIT Command

You can also edit a CMS maclib member by using the XEDIT command with the MEMBER option. For example, to edit the TEST2 member of TESTMAC MACLIB, enter:

```
xedit mylib testmac a1 (member test2
```

If the TEST2 member did not exist in TESTMAC MACLIB, this new member is added to the macro library. See “Adding MACLIB Members” on page 310 for more details.

Printing and Displaying MACLIB Members

To print or display members of a macro library, you may use the PRINT, TYPE, or MACLIST command.

Using PRINT and TYPE Commands

The PRINT and TYPE commands both accept the option MEMBER as a means of specifying a single MACLIB member or all the members. The format of these commands is similar.

For example, to print the TEST1 member of TESTMAC MACLIB, enter the following command:

```
print testmac maclib (member test1
```

Or, if you use the MEMBER option with an asterisk (*), all the members are displayed. For example, the following command displays all the members of TESTMAC maclib:

```
type testmac maclib (member *
```

One spool file is produced for each member printed. To print all the members of a library continuously without separator pages between them, issue the SPOOL PRINT CONT command. Then, if you want to return to printing files with separator pages between them, use the SPOOL PRINT NOCONT CLOSE command.

Using MACLIST Command

To print the TEST1 member of TESTMAC MACLIB from the MACLIST screen, type directly on the line that contains the TEST1 member:

```
print TEST1       266       5   TESTMAC    MACLIB    A1
```

Then, press Enter. See the MACLIST command in the *z/VM: CMS Commands and Utilities Reference* for more information about using symbols in MACLIST.

Another way to enter commands that make use of member names displayed is to move the first (or only) member you want the command to use to the current line. Then enter an EXECUTE command (in the form EXECUTE *lines command*) from the MACLIST command line. This method may be used on both display and typewriter terminals.

Extracting MACLIB Members

To extract a member from a macro library, you may use either the MOVEFILE and FILEDEF commands or the PUNCH command.

Using MOVEFILE and FILEDEF Commands

The MOVEFILE command with an appropriate FILEDEF extracts a member from a library. The MACLIB member you specify is copied directly from the MACLIB to your A-disk.

To copy a member from a given MACLIB onto your A-disk (for example, to make changes to it), issue a:

- File definition for the member name that is input to the MOVEFILE command
- File definition for the output file written to your A-disk
- MOVEFILE command.

Example 1

Suppose you want to make some changes to TEST5A DSECT in TESTMAC MACLIB. When you added TEST5A DSECT to TESTMAC MACLIB, you may have erased the source copy to save some disk space. So, the original is no longer available to you.

The following command sequence extracts the TEST5A DSECT from TESTMAC MACLIB and copies it to your A-disk with the file identifier of TEST5A COPY A1:

```
filedef inmove disk testmac maclib (member test5a
filedef outmove disk test5a copy a1
movefile
```

Now you can edit TEST5A COPY and make the changes you want. Then you can do a MACLIB REP to replace TEST5A in TESTMAC MACLIB.

Example 2

The MOVEFILE command in the preceding example is a simple application that makes use of the existing FILEDEFs. But with the PDS option, you can use MOVEFILE to extract every member of a macro library.

For example:

```
filedef test1 disk testmac maclib a
filedef macro disk
movefile test1 macro (pds
```

This sequence defines TESTMAC MACLIB as the input file for the MOVEFILE command and assigns a temporary logical name of TEST1 to the file. The second FILEDEF command identifies what the file type of the resulting file should be, MACRO, and where the file should be written, to your A-disk. The MOVEFILE command then causes TEST1 (that is, TESTMAC MACLIB A1) to be moved into separate files, each with a file type of MACRO.

Each member in this example has a file type of MACRO, including those with the original file type of COPY. You must rename those back to their original file type of COPY by using the CMS RENAME command.

Note: All CMS files you created by this method include the MACLIB delimiter statement / / as the last record in the file. So the first change you should make to a MACLIB member extracted in this way is to delete this / / delimiter record.

Using PUNCH Command

You can also extract a member from a macro library by using the PUNCH command. If you use the PUNCH command, first spool your virtual card punch to your own virtual reader:

```
cp spool punch to *
```

Then, to punch the macro library member to your virtual reader, enter:

```
punch testmac maclib (member get
```

To read it back onto disk, enter:

```
receive spoolid get macro
```

Setting MACLIST Defaults

When you issue the MACLIST command, you are placed in the XEDIT environment. Therefore, the default XEDIT macro, PROFMLST XEDIT, is executed. If you want to invoke a different XEDIT macro, you can specify the PROFILE option with the MACLIST command. For example, to invoke MACLIST with the MYMCLST XEDIT macro, enter:

```
maclist testmac (profile mymclst
```

You can do the same with the COMPACT and NOCOMPACT options of the MACLIST command.

If you are using an alternate profile most of the time, you may change the default profile with the DEFAULTS command. For example:

```
defaults set maclist profile mymclst
```

Issuing the DEFAULTS command with no options provides you with the status of defaults currently in effect.

Creating and Manipulating Text Libraries

A text library (TXTLIB file type) contains files with a file type of TEXT. These TEXT files 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.

TXTLIBs, like MACLIBs, have directories and members. The TXTLIB command creates a TXTLIB or changes the contents of a TXTLIB. The TXTLIB command reads the object files as it writes them into the library. It creates a directory entry for each entry point or CSECT name or file name if the FILENAME option is specified. The GLOBAL command defines the library for the loader program, and specifies the member name (the entry point) in the LOAD command. The TXTLIB command has a similar format as the MACLIB command, except that you cannot use the REP and COMP functions on the TXTLIB command.

Note: You should not use the OS/MVS STOW macro to create or modify TXTLIBs; only use the TXTLIB command to manipulate TXTLIBs. You should not use any OS macros to create or update TXTLIBs. OS partitioned data sets and CMS TXTLIBs interpret certain common fields differently. For this reason, OS macros, which were written to support OS conventions, do not create CMS libraries that can be used reliably.

Text libraries, like macro libraries, can reside in SFS directories. If you perform the GEN, ADD, DEL, or MAP functions of the TXTLIB command on a text library that resides in an SFS directory and the resulting library has no members, an empty text library is maintained on the SFS directory. This is done to preserve any authorities to the text file. You should use only the TXTLIB command to perform functions on an empty text library.

Each TEXT file contains at least one of each of the following types of records:

ESD
is an ***External Symbol Dictionary statement***. This is the first statement in the module (and therefore the first statement in each member of a TXTLIB). The ESD statement contains the name of the entry point (CSECT) of the module.

TXT
is a statement that contains the actual machine code of the program generated by the compiler or the assembler.

LDT
is a ***Loader Termination statement***. It contains data required by the loader program when the module is loaded into storage before execution or the creation of a nonrelocatable module.

The TEXT file cannot be executed directly because it is relocatable; the addresses are all relative to location zero. This is the standard form for all assembler and compiler output.

Using MVS/XA Linkage Editor Control Statements

You may add MVS/XA linkage editor control statements such as NAME, ALIAS, ENTRY, and SETSSI to a TEXT file (using XEDIT) before adding it to a TXTLIB. You must follow linkage editor conventions concerning format (column 1 must be blank) and placement within the TEXT file. Before adding the TEXT file to a TXTLIB, the control statements are processed as follows:

NAME Statement: A NAME statement causes the TXTLIB command to create the directory entry for the member using the specified name. Thereafter, when you want to load that member into storage or delete it from the TXTLIB you must refer to it by the name specified on the NAME statement.

Note: The FILename option overrides any name card found in a text file. The name card functions as before, but the specified file name becomes the member name in the TXTLIB. The name card is the only entry point within that member name of the TXTLIB. If a name card is not found in the text file and you specify the FILename option, the file's name is the member name. The first CSECT in the text file is the first entry point (the remaining entry points in the text file follow) within that member.

The loader does not use name cards to resolve entry points. It is important that the name on the name card be the same as the name on the CSECT or entry card. This ensures that the loader finds the correct text deck and loader tables (any external references) are resolved with the entry point. If the names differ, the loader loads the text deck based on the name card (or file name). However, the loader tables are set up according to entry or CSECT cards encountered during the load. Any external reference using the name from the name card is resolved as zeros.

ENTRY Statement: If you use an ENTRY statement, the entry point you specify is validated and checked for a duplicate. If the entry point name is valid and there are no duplicates in the TEXT file, the entry name is written in the LDT card. Otherwise, an error message is issued. When this member is loaded, execution begins at the entry point specified.

ALIAS Name: An entry is created in the directory for the ALIAS name you specify. A maximum of 64 alias names can be used in a single text deck. You may load the single member and execute it by referring to the alias name, but you cannot use the alias name as the object of V-type address constant (VCON) because the address of the member cannot be resolved.

SETSSI Card: TXTLIB command information you specify on the SETSSI card is written in bytes 26 through 33 of the LDT card.

All other MVS/XA linkage editor control statements and commands are ignored by the TXTLIB command and written into the TXTLIB member. When you attempt to load the member, the CMS loader flags these cards as not valid. These cards may be added as history information to a module if you specify the HIST option on the LOAD or INCLUDE commands and then issue a subsequent GENMOD command.

Creating a TXTLIB

The GEN function of the TXTLIB command generates a TXTLIB on your disk or directory. The TXTLIB command reads the object files as it writes them into the library and creates a directory entry. If you specify the FILENAME option, the member name in the text library is the file name of the text file used to create (add) this member. If you do not specify the FILENAME option, the member name in the text library is the entry point name or the CSECT name.

For example, suppose you have the following three text files:

- TESTPRG1 TEXT A1 with an entry point named TEST1
- TESTPRG2 TEXT A1 with an entry point named TEST2
- TESTPRG3 TEXT A1 with an entry point named TEST3

Enter the following command to create a text library:

```
txtlib gen testlib testprg1 testprg2 testprg3 (filename
```

Because you specified the FILENAME option, the member names of TESTLIB TXTLIB are the names of the text files (TESTPRG1, TESTPRG2, TESTPRG3). The file names, as well as the entry point names, are put in the directory.

Now, enter the following command to create a text library:

```
txtlib gen testlib testprg1 testprg2 testprg3
```

Because you did not specify the FILENAME option, the member names of TESTLIB TXTLIB are the entry point names (TEST1, TEST2, TEST3) of the text files. You must use these entry point names to reference the specific TXTLIB members. TEST1, TEST2, and TEST3 are the names put in the directory also.

Note: The total number of members in any given TXTLIB cannot exceed 6000. An error message is displayed when this number is reached. When processing terminates, the TXTLIB created includes all the text files entered up to, but not including, the one that caused the overflow.

The total number of entry points in each member cannot exceed 4048. An error message is displayed when this limit is reached and processing has begun on a new file. When processing terminates, the TXTLIB created includes all the text files except the one that caused the overflow.

Examining the Contents of a TXTLIB

The MAP function of the TXTLIB command lists information about the members of a TXTLIB. This information includes:

- Names of the members of the TXTLIB
- Location of the members in the TXTLIB
- Number of entries.

You can obtain this information as a:

- File on your A-disk (the DISK option)

  The DISK option creates a file with the file type MAP. The file name is the same as the TXTLIB being mapped. If a MAP file already exists for the specified TXTLIB, it is erased and a new MAP file is created.
- Spooled printer file (the PRINT option)

- Display on your terminal (the TERM option).

Example: To display the information about the members in TESTLIB TXTLIB created earlier, enter the following command:

```
TXTLIB MAP TESTLIB (TERM
```

If you created TESTLIB TXTLIB without the FILENAME option, the following is displayed:

```
  ENTRY          INDEX
  TEST1              2
  TEST2             39
  TEST3             76
            3 ENTRIES IN LIBRARY
```

TEST1, TEST2, and TEST3 are entry point names of the text files.

Adding TXTLIB Members

The ADD function of the TXTLIB command adds a text file to a text library. If you specify the FILENAME option, the member name in the text library is the file name of the text file. If you do not specify the FILENAME option, the member name in the text library is the entry point name or the CSECT name.

Deletions must be made on the member names. CMS commands will treat entry point name(s) the same as member name(s). They may be different from the CMS file name from which they originated depending on whether you specified the FILENAME option.

Suppose you want to add TESTPRG4 TEXT, with a CSECT named TEST4, to the text library, TESTLIB. If you specify the FILENAME option when you enter the TXTLIB command:

```
txtlib add testlib testprg4 (filename
```

TESTLIB TXTLIB has a new member called TESTPROG with TEST4 as an entry point in that member. TESTPRG4, as well as TEST4, are added to the directory.

If you do not specify the FILENAME option when you enter the TXTLIB command:

```
txtlib add testlib testprg4
```

TESTPROG TXTLIB has a new member called TEST4. TEST4 is added to the directory.

Deleting TXTLIB Members

The DEL function of the TXTLIB command deletes members of a text library. If the TXTLIB contains more than one member with the same name, only the first member is deleted. You must delete the member name, which may not be the name of the text file used to create this member. It depends on whether you used the FILENAME option when the member was added.

Any attempt to delete a specific alias or entry point within a member results in a NOT FOUND message.

For example, to delete the member TEST3 from TEXTLIB, enter:

```
txtlib del testlib test3
```

Replacing TXTLIB Members

The TXTLIB command does not have a REP function. To replace a member in a TXTLIB, use the DEL function followed by the ADD function.

For example, suppose you want to replace the member TEST3 in TEXTLIB with TESTPRG5 TEXT, with a CSECT named TEST5. First delete the TEST3 member by entering:

```
txtlib del testlib test3
```

Remember that you must delete the member name, which may not be the name of the text file used to create this member name. It depends on whether you used the FILENAME option when the member was added.

Now, add TESTPRG5 TEXT to TESTLIB TXTLIB by entering one of the following commands, depending on what you want the member name to be:

```
txtlib add testlib testprg5 (filename
```

or

```
txtlib add testlib testprg5
```

If you specify the FILENAME option, the member name is TESTPRG5. If you do not specify the FILENAME option, the member name is TEST5.

Printing and Displaying TXTLIB Members

The PRINT command with the MEMBER option prints the contents of a text library or prints a specific entry point or member of a text library. The TYPE command with the MEMBER option displays the contents of a text library or displays a specific entry point or member of a text library. These CMS commands will search all member names and entry point names in sequential order.

For example, to print the TEST2 member of TESTLIB TXTLIB, enter:

```
print testlib txtlib (member test2
```

One spool file is produced for each member printed. To print all the members of a library continuously without separator pages between them, use the SPOOL PRINT CONT command. Then, if you want to return to printing files with separator pages between them, use the SPOOL PRINT NOCONT CLOSE command.

To display all the members of TESTLIB TXTLIB, enter the following command:

```
type testlib txtlib (member *
```

Extracting TXTLIB Members

The PUNCH command extracts a member from a text library. This CMS command will search all member names and entry point names in sequential order. If you use the PUNCH command, first spool your virtual card punch to your own virtual reader by entering:

```
cp spool punch to *
```

Then, to punch the TXTLIB file to your virtual reader, enter:

```
punch testlib txtlib (member get
```

To read it back onto disk, enter:

```
receive spoolid get text
```

Creating and Manipulating Load Libraries

A load library, LOADLIB, is another type of library also available. LOADLIBs, like MACLIBs and TXTLIBs, are in CMS simulated partitioned data set format.

You create and manipulate LOADLIBs differently than you would MACLIBs and TXTLIBs. To create a LOADLIB or a LOADLIB member, use the LKED command. To manipulate load libraries, use the LOADLIB command. To execute a LOADLIB member, use the OSRUN command.

Creating LOADLIBs Using the LKED Command

The LKED command creates a CMS LOADLIB or a member of a LOADLIB from a TEXT file. For example, to create a LOADLIB named PROG025 LOADLIB from the PROG025 TEXT file, enter:

```
lked prog025
```

This LOADLIB contains an executable load module that you can run using the OSRUN command. See "Using the LKED and OSRUN Commands" on page 58 for information on executing programs stored as a member of a LOADLIB.

The LKED command produces two permanent files on your A-disk. The file name of both files is the name specified on the LKED command. One file contains the load module(s) created by the linkage editor. It is given the file type LOADLIB. The other file is the printed output from the linkage editor. It is given the file type LKEDIT.

The LKED command lets you specify many different options. CMS does not use all of the options. The CMS-related options are: TERM, NOTERM, PRINT, DISK, NOPRINT, AMODE, RMODE, NAME, and LIBE. The remaining options are: LET, NE, OL, RENT, REUS, REFR, OVLY, XREF, MAP, LIST, NCAL, XCAL, SIZE, and ALIGN2. See the *z/VM: CMS Commands and Utilities Reference* for a description of these options.

Manipulating LOADLIBs Using the LOADLIB Command

The LOADLIB command maintains CMS LOADLIBs. The LOADLIB command lists the members of a LOADLIB, copies members from one LOADLIB to another, merges complete LOADLIBs, or compresses a LOADLIB. See the *z/VM: CMS Commands and Utilities Reference* for information on the LOADLIB command.

Creating Callable Services Libraries

The CSLGEN command is used to create a callable services library (CSL). CSLGEN can create CSLs which reside on DASD, directory, or within a segment. For example:

```
CSLGEN DASD MYLIB FROM YOURLIB
```

will create a CSL named MYLIB from a build list called YOURLIB CSLCNTRL. While:

```
CSLGEN SEG MYLIB FROM YOURLIB
```

will create the same CSL ready to be included in a segment. For more information on placing CSLs in segments see the *z/VM: CMS Commands and Utilities Reference*.

When the CSL contains direct call routines, a TXTLIB file with the same file name as the CSL is created as well. This TXTLIB contains entries known as call routing code segments. These code segments are to be linked to the application program so that the CSL routine call is routed to the proper address in storage.

The main input of CSLGEN is a build list which by default has the file type of CSLCNTRL. The CSLCNTRL file provides CSLGEN with a list of all of the routines to be included in the CSL as well as what TEXT deck and TEMPLATE file to use for each. Other optional attributes can be specified for each routine as well. For a more complete description of the CSLCNTRL file and how to create one see the *z/VM: CMS Application Development Guide for Assembler*.

Using Callable Services Libraries

CMS includes two callable services libraries named VMLIB and VMMTLIB. VMLIB contains CSL routines that:

- Call CMS file system management functions (CMS file pool and minidisk I/O)
- Call CMS file pool administration functions
- Take advantage of CMS's data integrity capabilities

- Access the current generation of REXX variables
- Interface with the z/VM command environment through a REXX exec
- Invoke the CMS Extract/Replace facility, which lets applications obtain or modify selected system information without release or z/VM system dependencies
- Call data space services available on CMS
- Call program-to-program communications functions using 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
- Provide CMS file pool exits

VMMTLIB contains CSL routines that:

- Call CMS application multitasking functions
- Call OpenExtensions services
- Get the value set for the workstation display address

DFSMS/VM provides two additional callable services libraries. FSMPPSI contains Removable Media Services (RMS) Tape Library Dataserver interface routines. FSMPSI contains DFSMS/VM installation-wide exit routines.

Your applications can call all these CSL routines, much like subroutines, to perform z/VM services without writing unique assembler subroutines. These calls are not resolved until the call is made (as opposed to when the program is linked or loaded). This lets you make changes to a CSL routine without having to relink the routine to the application program, recompile the application, or modify any of the program's source statements.

The following books provide information about the CSL routines listed above:

- *z/VM: CMS Callable Services Reference* describes the basic set of routines in VMLIB. It also describes the VMMTLIB routine that gets the value set for the workstation display address.
- *Common Programming Interface Communications Reference (https://publibfp.dhe.ibm.com/epubs/pdf/c2643999.pdf)* describes the SAA CPI Communications routines.
- *Common Programming Interface Resource Recovery Reference* describes the SAA resource recovery routines.
- *z/VM: CMS File Pool Planning, Administration, and Operation* describes the CMS file pool exit routines.
- *z/VM: CMS Application Multitasking* describes the CMS application multitasking routines.
- *z/VM: OpenExtensions Callable Services Reference* describes the OpenExtensions callable services.
- *z/VM: DFSMS/VM Removable Media Services* describes the RMS Tape Library Dataserver interface routines.
- *z/VM: DFSMS/VM Customization* describes the DFSMS/VM installation-wide exit routines.

You can also create your own CSL routines and build your own CSL. You can use these routines the same way you would use the CSL routines supplied with VMLIB and VMMTLIB. See *z/VM: CMS Application Development Guide for Assembler* for information on creating your own CSL routines and building a CSL.

The following information describes how to load a CSL, manipulate CSL routines, and invoke a CSL routine.

Making CSLs Available for Use

The GLOBAL command with the CSLLIB operand manipulates the search order of the libraries that the RTNLOAD command uses to locate CSL routines. If you do not make the library specifically available with the GLOBAL command, you must load a routine from the library by specifying the library directly with the FROM option on RTNLOAD.

Note: VMLIB and VMMTLIB are always implicitly in the search order. If the GLOBAL CSLLIB command is issued without any library name, VMLIB and VMMTLIB are still searched.

See the *z/VM: CMS Commands and Utilities Reference* for a complete description of the GLOBAL and RTNLOAD commands.

Loading or Dropping a CSL Routine

The RTNLOAD command loads a CSL routine from a library that is stored in a logical saved segment, on disk, or in the CMS nucleus.

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 contained within the CMS nucleus and is automatically loaded during CMS initialization before SYSPROF EXEC is run.

You can tailor the SYSPROF EXEC, using the RTNLOAD command, to make your own callable services library available to many users.

The RTNDROP command drops a CSL routine that was loaded with RTNLOAD. VMMTLIB routines cannot be dropped via the RTNDROP command, although calls to them can be intercepted by routines that are loaded by the RTNLOAD command.

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. You can RTNLOAD and RTNDROP these library subgroups with one command. Such groupings can help save storage and improve call time performance.

See the *z/VM: CMS Commands and Utilities Reference* for a complete description of the RTNLOAD and RTNDROP commands.

Getting Information about Routines in a Library

You can use various commands to get more information about your CSL routines:

- The RTNSTATE command verifies that a routine is loaded.
- The CSLLIST command displays information of the contents of a callable services library.
- The CSLMAP command displays information about currently loaded CSL routines in an interactive environment.
- The QUERY CSLLIB command displays names of the CSL libraries in the search order. The QUERY LIBRARY command displays names of all library files.
- The RTNMAP command displays information about the CSL routines that are currently loaded.

See the *z/VM: CMS Commands and Utilities Reference* for a complete description of the RTNMAP, RTNSTATE, CSLLIST, QUERY CSLLIB, and QUERY LIBRARY commands.

Programming Language Binding Files

For certain groups of CSL routines, z/VM provides programming language binding files that define entry points, declare external functions, and define constants, such as return codes and reason codes. You must include these binding files in your program before you invoke the CSL routines.

The following types of CSL routines use binding files:

- CMS application multitasking functions—see *z/VM: CMS Application Multitasking* for more information.
- OpenExtensions callable services—see the *z/VM: OpenExtensions Callable Services Reference* for more information.
- VMLIB routines that use names longer than eight characters—see the *z/VM: CMS Callable Services Reference* for more information.

Invoking a CSL Routine

To invoke a CSL routine, whether it is a routine you created or one already in one of the system libraries, you must use one of the following methods.

Note:

1. To invoke CPI Communications (also known as SAA communications interface) routines and SAA resource recovery (also known as CPI resource recovery) routines, you must use a different call format than the one described in this chapter. For more information on CPI Communications, see Chapter 33, "Understanding CPI Communications," on page 493.
2. To invoke CMS multitasking routines, you must use the call formats described in the *z/VM: CMS Application Multitasking*.

Direct Call
A call routing code segment generated by CSLGEN for the CSL routine directs the call to the CSL routine. This call interface can be used only with direct call CSL routines. It is much faster than DMSCSL and is usable by high-level languages, unlike CSLFPI. Performance is comparable to that of the "fastpath" interface available to assembler programs.

The call routing code segments are held in a TXTLIB file with the same file name as that of the CSLLIB, CSLSEG, or TEXT file created by CSLGEN. The individual call routing code segment finds the proper CSL routine version using a path assigned to the routine name on the ROUTINE line of the CSLCNTRL file used to build the library.

Use the GLOBAL TXTLIB command to access the CSL TXTLIB files needed by your application program. Before calling a CSL routine, the CALL Router code segment must be linked to the program using the LOAD/INCLUDE command or the LKED control statement. Additional language-specific statements may be necessary so that language compilers can provide the proper assembler interface. See the *z/VM: CMS Application Development Guide for Assembler* for additional information on direct call CSL routines.

Call DMSCSL
Call DMSCSL passes control to another module that searches for the specified routine, converts the parameter list, and invokes the CSL routine. Before calling a CSL routine, DMSCSL must be linked to the program using the LOAD/INCLUDE command or the LKED control statement. Additional language-specific statements may be necessary so that language compilers can provide the proper assembler interface.

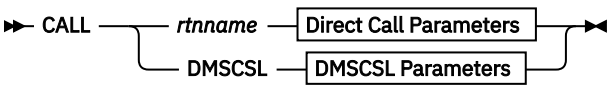
CSLFPI Macro
CSLFPI is used by assembler programs to invoke CSL programs using a 'fast path'. A program should use a fast path when it calls the same CSL routine several times and optimum performance is important. See "Invoking CSL Routines Frequently from Assembler Programs" on page 326 for more information on CSLFPI.

REXX
REXX can call a CSL routine as a subroutine or as a function. See the *z/VM: REXX/VM Reference* for more information on calling CSL routines from REXX.

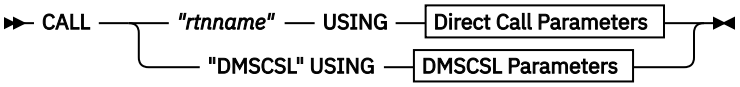
Calling Formats

Here are the calling formats for all the supported languages:

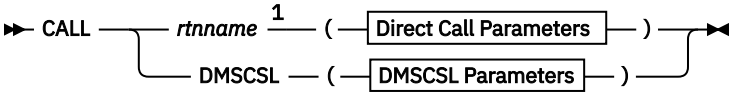
General Format



VS COBOL II



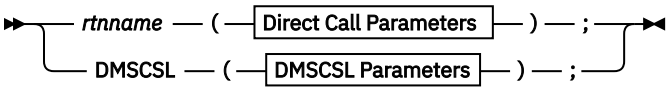
VS FORTRAN



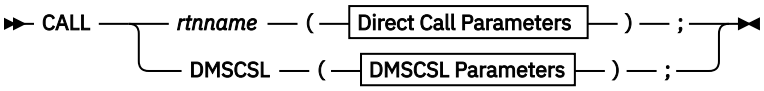
Notes:

[1] VS FORTRAN cannot use the longer-style routine names, such as StackBufferCreate, to make direct calls. It must use the shorter name, such as DMSSTKC.

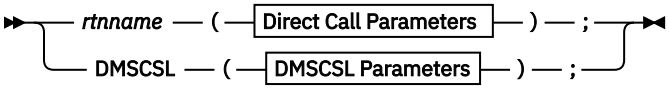
VS Pascal



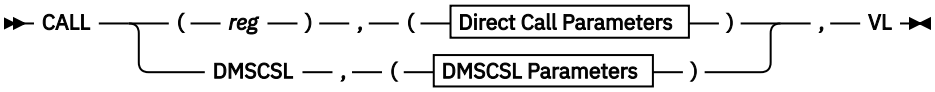
PL/I



C



Assembler



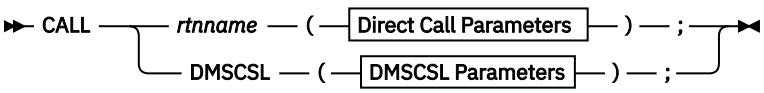
Note:

1. When the long form of a routine's name is used in a direct call, the routine's address must be passed in a register, for example:

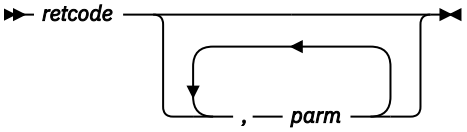
```
L     R15,=A(STACKQUERY)
CALL  (15),(RC,RE,BUFNUM,LINES,HIGH),VL
```

2. For the Assembler language, addresses used with CSL routines are 32-bit fields. The high-order bit is not used for addressing and must be zero, except that it must be set to one when it designates the end of a parameter list. Specifying VLon the routine call sets the high-order bit to 1. If you build the parameter list yourself and provide only the address of the list in the routine call, you must set the high-order bit of the last address in the list (see the information on programming language binding files in the *z/VM: CMS Callable Services Reference*).

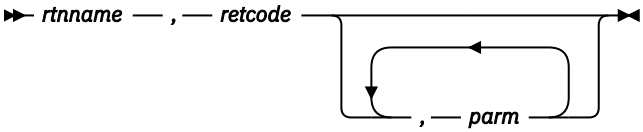
Ada



Direct Call Parameters

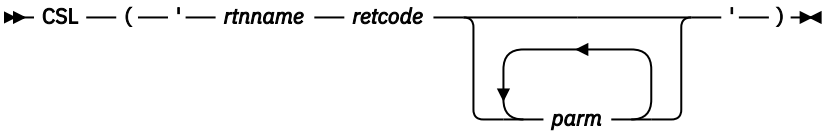


DMSCSL Parameters

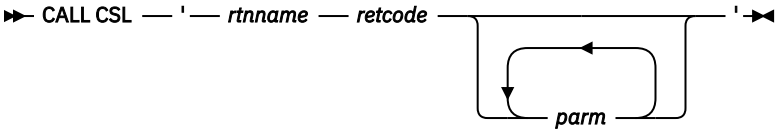


Calling CSL routines is different in REXX/VM. Routines can be called as functions or they can be called as routines with the CALL instruction or the ADDRESS OPENVM instruction:

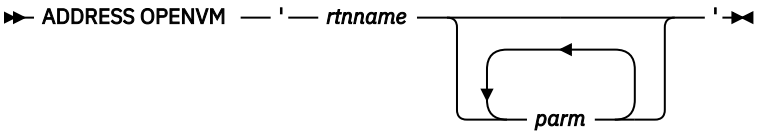
REXX function call



REXX CALL instruction



REXX ADDRESS OPENVM instruction



Note: OPENVM type CSL routines can be called from REXX *only* by using the ADDRESS OPENVM interface. OPENVM routines may not follow the usual CSL conventions, such as providing return and reason codes as the first two parameters. For more information about the ADDRESS OPENVM interface, see the *z/VM: REXX/VM Reference*. To determine if a CSL routine is an OPENVM routine, you can use the CSLMAP or CSLLIST command. For information about these commands, see the *z/VM: CMS Commands and Utilities Reference*.

Parameters

rtnname
is the name of a variable that contains the name of the CSL routine being called. This is only a parameter for the DMSCSL call and REXX.

retcode
holds the return code from the CSL routine.

parm
are the parameters passed to the CSL routine.

For information about specific CSL routines in VMLIB, see the *z/VM: CMS Callable Services Reference*. For information about specific CSL routines in VMMTLIB (except the OpenExtensions routines), see the *z/VM: CMS Application Multitasking*.

For information about the OpenExtensions routines in VMMTLIB, see the *z/VM: OpenExtensions Callable Services Reference*.

Example

Suppose your FORTRAN program wants to find the access mode of the first read/only CMS disk. It can do this using the extract function of the DMSERP routine. DMSERP is a CSL routine that obtains or updates specific system information.

First, your program should declare and initialize the variables to be used. For example, FUNCT, the variable containing the function of RNAME, is set to EXTRACT. Then, use the following format of the CSL routine to extract the access mode of the first read/only CMS disk.

```
CALL DMSCSL (rtnname, RETCODE, FUNCT, NUMARGS, INFONAME, BUFFER,
             DATATYP, BUFLEN, FLAGS, SRCHTYP, TOKEN, SARGNAM, SARGVAL,
             SVALTYP, SVALLEN, SARGTYP)
```

Return Codes

For a list of the return codes you can receive from calling a CSL routine, see the *z/VM: CMS Callable Services Reference*. If the call to the CSL routine was made from a REXX program, additional return codes are generated. See the *z/VM: REXX/VM Reference* for details on these return codes.

Invoking CSL Routines Frequently from Assembler Programs

If an assembler application program is going to frequently invoke the same CSL routine, you can increase the performance of your program by using a *fastpath* interface with the CSLFPI macro.

You implement the fast path using these four steps:

1. Make an area that contains information about the CSL routine and its parameters. This information includes the location of the routine and the location and size of the routine's parameters. You can build this area using CSLFPI TYPE=AREA, or define the area and map it onto other storage using CSLFPI TYPE=DSECT.
2. Give initial values for the CSL routine and its parameters using CSLFPI TYPE=INIT or CSLFPI TYPE=INITD. INIT should be used if you actually built the area in step 1 (using TYPE=AREA); INITD should be used if you mapped the area onto unformatted storage in step 1 (using TYPE=DSECT).
3. Set or change values for the address and length values for a CSL routine's parameters using CSLFPI TYPE=SET. (This step is not required unless you wish to change the values originally established when the fast path area was initialized.)
4. Once you have defined the necessary information for a CSL routine and its parameters, you invoke the CSL routine using CSLFPI TYPE=CALL. You can also set or change parameter address and length information on this macro call.

If you use the VM Data Spaces facility, see Chapter 15, “Using Data Spaces,” on page 217 for information on calling CSL routines from access register mode using the CSLFPI macro.

Examples of each of these steps are shown later. See the *z/VM: CMS Macros and Functions Reference* for detailed information on the CSLFPI macro.

1. Setting Up the Fast Path Area

   The following program portions show three examples of using CSLFPI TYPE=AREA and CSLFPI TYPE=DSECT.

   - This builds an area for three parameters. It gives names for all three. FP1 is the name used to reference the fast path area.

     ```
     FP1      CSLFPI TYPE=AREA,
                    PARMS=((RETR),(REAS),(P1))
     ```

   - This defines a mapping for unformatted storage. (The storage would have to be previously allocated, such as by a call to the CMSSTOR macro.) It just names four parameters.

     ```
     FP2      CSLFPI TYPE=DSECT,
                    PARMS=((RETR),(REAS),(P2),(P3))
     ```

   - This defines a mapping within a DSECT. (The storage would have to be previously allocated, possibly passed from another module.) The names of four parameters are specified. In addition, this defines CSLROUT3 as the CSL routine being called.

```
PLACE    DSECT
FP3      CSLFPI TYPE=AREA,SERVICE=CSLROUT3,
               PARMS((RETR),(REAS),(P4),(P5))
```

2. Initializing the Fast Path Area

   The following program portions show two examples of using CSLFPI TYPE=INIT and CSLFPI TYPE=INITD. These examples are based on the previous TYPE=AREA/DSECT examples:

   - This defines CSLROUT1 as the CSL routine being called, as well as the addresses for parameters P1, RETR, and REAS.

     ```
              CSLFPI TYPE=INIT,AREA=FP1,SERVICE=CSLROUT1,
                    PARMS=((RETR,RETURN),(REAS,REASON),(P1,PARM1))

     RETURN   DC    F'0'
     REASON   DC    F'0'
     PARM1    DC    C'This is parameter 1'
     ```

   - This defines CSLROUT2 as the CSL routine being called, as well as the addresses for parameters RETR, REAS, and P2. Note that INITD must be used because the area was mapped onto unformatted storage with TYPE=DSECT.

     ```
     INIT2    CSLFPI TYPE=INITD,AREA=FP2,SERVICE=CSLROUT2,
                    PARMS=((RETR,RETURN),(REAS,REASON),
                    (P2,PARM2))

     RETURN   DC    F'0'
     REASON   DC    F'0'
     PARM2    DC    F'222'
     ```

3. Modifying Fast Path Area Values

   The following program portions show two examples of CSLFPI TYPE=SET. These examples are based on the previous TYPE=AREA/DSECT and TYPE=INIT examples:

   - This sets a length for parameter P1:

     ```
              CSLFPI TYPE=SET,AREA=FP1,
                    PARMS=((P1,,L'PARM1)
     ```

   - This sets the length and address for parameters P4 and P5:

     ```
              CSLFPI TYPE=SET,AREA=FP3,
                    PARMS=((RETR,RETURN),(REAS,REASON),
                    (P4,PARM4,L'PARM4),(P5,PARM5,4))

     RETURN   DC    F'0'
     REASON   DC    F'0'
     PARM4    DC    C'This is parameter 4'
     PARM5    DC    F'555'
     ```

   Note:

   a. The values and lengths of parameters should not be assigned using a TYPE=AREA statement if directly callable CSL routines may be called.

   b. A length must be set for a parameter when the parameter is defined in the CSL routine's template as being type FCHR and having a length of zero.

   c. The length attributes for parameters which are not of type FCHR, with length zero, are set automatically at TYPE=INIT/INITD time for CSL routines that are not directly callable. They are ignored when initializing for a directly callable routine.

   d. The length of a parameter, which is followed immediately by an LEN type parameter, is not specified as shown above. The length is supplied as the value of the LEN type parameter which follows it.

4. Invoking a CSL Routine

The following program portions show two examples of CSLFPI TYPE=CALL. These examples are based on the previous examples.

- This invokes the CSL routine named CSLROUT1, whose fast path area was named FP1:

```
        CSLFPI TYPE=CALL,AREA=FP1
```

- This invokes the CSL routine named CSLROUT3, whose fast path area was named FP3. Note that a new address and length is being supplied for P4.

```
        LA    R3,PARM6
        LA    R4,L'PARM6
        CSLFPI TYPE=CALL,AREA=FP3,
              PARMS=((P4,(R3),(R4))
```

Using ISPF/PDF Libraries

Application programmers often work in groups to develop application programs. In many cases, a programmer is a specialist in certain areas of application programming, such as writing assembler language subroutines to be called by programs written in high-level languages. You may be responsible for creating certain file structures for use in multiple applications.

For example, you may be asked to create and maintain the Data Division statements that define certain file structures to be used by a number of COBOL programs, written by other programmers. Or, you may need to write certain FORTRAN subprograms to be called by main programs during processing.

To help share source and object code, the Interactive System Productivity Facility (ISPF) has a companion product called Program Development Facility (PDF) that you can use to create and maintain libraries of shared source code, object code, data, or documentation. These libraries may be sets of CMS files, MACLIBs, or TXTLIBs. They are identified by project name, group name, and type of information in the library.

An ISPF/PDF library is a collection of code or data units called members. Each library generally contains members with the same type of information. For example, all the members of one library may consist of assembler source code. Another could contain COBOL Data Division definitions, or documentation files written in SCRIPT. ISPF/PDF libraries are maintained internally as CMS files. Each library may consist of a set of CMS sequential files, or it may be a MACLIB or TXTLIB. The particular organization is designated when the library is specified to PDF using the file utility (UTILITIES on the first screen and FILE on the second screen).

Each ISPF/PDF library is identified by:

Project name
is the common identifier for all libraries belonging to the same project.

Group name
is the identifier for a particular set of libraries.

Type
is the identifier for the type of information in the library.

PDF usually represents these characteristics the same way an OS partitioned data set is represented—you join them with a period. For example, if your project name is PERSONNEL, the group name is TESTLIB, and the information type is COBOL, the library would be specified as:

```
PERSONNEL.TESTLIB.COBOL
```

Most projects use a hierarchy of related libraries to maintain effective version control over the programming development process and to reduce contention in library usage. For example, there may be three levels of library for a given project: a master library for production, a test library, and multiple development libraries. The master library designator could be PRODLIB, the test library TESTLIB, and the development library DEVLIB. The development library could also be given the name of the CMS user who owns the particular library.

For the PERSONNEL project, you could have the following library names:

PERSONNEL.PRODLIB.COBOL
PERSONNEL.TESTLIB.COBOL
PERSONNEL.DEVLIB.COBOL

Each library is uniquely named. This gives great flexibility in accessing various members contained in them.

Specifying ISPF/PDF Libraries and Their Members

To specify a member of an ISPF/PDF library, you must enter a project name, group name, type qualifier, and member name. Each of these items may contain up to eight alphanumeric characters. For the project name, group name, and type name, the first character must be alphabetic; for a member name, the name must follow CMS file name naming conventions. PDF automatically issues the appropriate LINK and ACCESS commands necessary to access the minidisk on which the library resides.

PDF panels prompt you for each component of the library identification as follows:

```
ISPF LIBRARY:
    PROJECT ===>
    GROUP   ===>
    TYPE    ===>
    MEMBER  ===>
```

For example, to gain access to a member called TESTPROG, residing in the PERSONNEL.DEVLIB.COBOL library, you would respond to the PDF panel prompts as follows:

```
ISPF LIBRARY:
    PROJECT ===> personnel
    GROUP   ===> devlib
    TYPE    ===> cobol
    MEMBER  ===> testprog
```

If you do not specify the member name, PDF displays a list of members of the library, which you can browse before selecting a specific member. Member lists are provided for PDF functions. Some of these functions are BROWSE (to examine a file), EDIT (to make changes to a file), MOVE, and COPY.

Guidelines for Library Specifications

You must specify each ISPF library with the ISPF/PDF file utility (UTILITIES on the first screen and FILE on the second screen) before it can be used. The name of the library along with the following information must be specified:

ISPF/PDF Library Attributes
Organization, record format, and record length.

ISPF/PDF LIBRARY Location
Owner's ID and device address.

Link Access Mode
For update (write and multiwrite, among others). See *ISPF/PDF Services* for more information.

File type
For organization S (set of files).

File name
For organization M or T (MACLIB or TXTLIB)

An ISPF/PDF library takes one of three forms:

S
is a set of CMS sequential files, all with the same file type. The CMS file names are the same as the ISPF/PDF library member names. The CMS file type can be anything that uniquely identifies the set of files on a minidisk, such as COBOL, DATA, or TEXT.

M
is a CMS MACLIB, with a file name that uniquely identifies the MACLIB on the disk. The member names in the MACLIB are the same as the ISPF library member names.

T
is a CMS TXTLIB, with a file name that uniquely identifies the TXTLIB on the disk. The member names in the TXTLIB are the same as the ISPF library member names.

ISPF/PDF Library Record Format and Length

Libraries with an organization of M or T must have a record format of F (for fixed-length records) and a record length of 80. Libraries with S organizations may have a record format of F or V (for variable-length) with record lengths from 1 to 32,767 bytes. (However, the PDF editor can only process records that are longer than 9 bytes and shorter than 256.)

Location of ISPF/PDF Libraries

Each ISPF library must be completely contained on one minidisk. You specify this with the user ID of the owner, and the virtual address of the device on which the library resides.

You can have more than one ISPF/PDF library on the same minidisk. ISPF/PDF libraries can also exist on the same minidisk with other CMS files that are not ISPF/PDF libraries. Usually, the lowest level libraries in a project (the DEVLIBs in our example) are private libraries, owned by the principal or only user. These should have an organization of S to eliminate the need for compressions. Higher level libraries are usually common libraries accessed for reading by anyone on the project, but maintained by one designated individual.

For example, if your responsibility is to maintain test data for a given project, you would have write access to the PERSONNEL.TESTLIB.DATA library. Everyone else on the project would only have read access. This kind of restriction helps protect the integrity of the data. It helps ensure that everyone is using the same files.

If you want to protect higher level libraries against unauthorized access by those outside the project, minidisks on which they reside can be protected with read passwords. You can, for example, assign the same read password to all minidisks containing libraries for the PERSONNEL project. This lets people working on the project access any library, but prevents those outside the project from gaining access.

Concatenating ISPF/PDF Libraries

PDF lets you specify up to four libraries during source editing, compilation, assembly, or SCRIPT/VS processing (plus additional MACLIBs for compilations and assemblies). Generally, the lowest level library is specified ahead of the next higher level library, and so on, in bottom-to-top order. The following example shows how you could specify three libraries using the PDF library (member) specification panel:

```
ISPF LIBRARY:
    PROJECT ===> personnel
    GROUP   ===> devlib    ===> testlib  ===> prodlib  ===>
    TYPE    ===> cobol
    MEMBER  ===> testprog
```

In this example, three libraries are specified in this order for TESTPROG COBOL:

PERSONNEL.DEVLIB.COBOL
PERSONNEL.TESTLIB.COBOL
PERSONNEL.PRODLIB.COBOL

Specifying libraries this way during editing lets you copy members to your development library. Use the specification sequence to search the libraries for the member you want to edit. The edited member is saved in your development library (the first library in the concatenation sequence), while the unchanged version remains in the test or master library. When you have finished testing the new version, you can promote it to a higher level library using the move/copy utility, PDF option (UTILITIES on the first screen and MOVE/COPY on the second screen).

Library concatenation during language processing makes it easy to include source segments using INCLUDE or COPY statements (or SCRIPT imbed controls). You can debug new or modified programs without altering the contents of the test or master libraries. The output from a compilation or assembly (object module) is stored in the lowest level TEXT library (the first library in the concatenation sequence).

ISPF/PDF Library Statistics

When a list of library members is displayed (for example, when you leave the MEMBER field blank on the PDF library selection panel), various statistics associated with each member are displayed, including:

- Name of the member
- Version number
- Modification level
- Creation date
- Date last modified
- Size.

These statistics help you keep track of files. Next to the name of the library member there's a blank field that you can use to SELECT a member for editing, browsing, or other PDF functions. You do this by placing the letter S in the blank letter field.

See the *ISPF/PDF Services* for additional information on ISPF/PDF libraries and the PDF functions.

Chapter 20. Using Execs

This chapter describes:

- The different types of execs
- XEDIT macros
- Special execs you can write and use (PROFILE EXEC and CMS EXEC)
- How to use the FILEDEF and GLOBAL commands in an exec
- How to use execs to create prototypes for applications.

An ***exec*** is a file of statements that are executed when you enter a single statement. You often need to perform a sequence of CMS and CP commands, for example, when compiling and link editing a source program. You can group this sequence of commands in an EXEC file and control the execution of these statements by using exec language statements.

In its simplest form, an EXEC file may contain only one record. In its most complex form, it can contain thousands of records and resemble a complete program written in a high-level programming language. An exec can contain CMS commands, CP commands, or exec statements. Each exec language has its own set of statements and language syntax. You can also call an exec from within an exec.

Execs must have a file type of EXEC. CMS first searches for an exec in storage with the specified file name and a file type of EXEC. If the exec is not found, CMS searches for a file with the specified file name and a file type EXEC on any currently accessed directory or disk.

There are four types of execs:

- Restructured Extended Executor (REXX) execs
- EXEC 2 execs
- CMS EXEC execs
- Alternate format execs.

Restructured Extended Executor Language

Restructured Extended Executor (REXX) language is a general-purpose, high-level language, not unlike PL/I, which is especially suited for prototyping and personal computing as well as handling exec command procedures. REXX is a free-format language that can be coded to emphasize its structure, making it easier to read.

Although REXX is easy to use, REXX programs are executed using the REXX/VM interpreter; thus, it tends to use more computer time than an equivalent compiled language.

The clause length maximum has been increased. It was 500 characters; now the actual limit is the amount of storage that can be obtained on a single request.

Note: Throughout this chapter, the term REXX/VM interpreter is called *interpreter*.

Sample REXX Language Program

The following program illustrates some of the REXX language statements:

```
/* The first line of a REXX exec must always be a comment.
   The comments can span more than one line.            */
    credits = 0
    do until credits > 5
    a = random(1,9)
    b = random(1,9)
    say "What is "  a "plus"  b  "?"
    pull answer  /* Place user's reply into answer */
    if answer = a + b
    then do
```

```
          credits = credits + 1; say "Correct."
          say "Your score is" credits
          end
     else
          say a "+" b "is" a+b
     end
     exit
```

This program repeatedly asks for the sum of two random numbers until it has accumulated six correct answers. The following describes the sequence of execution:

- The comment delimiter `/*` on the first line indicates to CMS that this is a REXX program. This causes CMS to call the interpreter. The last comment line must end with `*/`.
- A value of 0 is assigned to the variable `credits`.
- The lines from `do until` to the second `end` are repeatedly executed as long as the value of `credits` does not exceed 5.
- Variables a and b are assigned random values in the range 1 to 9. The term `random(1,9)` is a built-in function; its arguments are the desired range in which the random number is to be generated. The REXX language has over fifty built-in functions. They are listed in *z/VM: REXX/VM Reference*.
- The instruction `say` writes the values of a and b to the console along with the literals "`What is`", "`plus`", and "`?`" in the order specified. One space is automatically inserted between separate literals or variables or both. If more than one space is required, it must be incorporated into a literal (as, for example, in "`What is `").
- The instruction `pull` accepts the console reply into the variable `answer`. Comments in the REXX language can be included on the same lines as program statements.

  There are two forms of the PULL instruction:

  – The form PARSE UPPER PULL, which is normally abbreviated to PULL, translates everything read from the keyboard to upper case in the program.
  – The form PARSE PULL should be used if everything is required as is, without any translation.
- In the statement `if answer = a + b`, the item to the right of the = sign can be an expression, in this case `a + b`.
- When the test is true, more than one statement is to be executed. The `do...end` delimits these statements. If only one statement is to be executed, the delimiters are not required.
- So far there has been only one REXX language statement per line. A line-end is considered to be an implied delimiter. However, if more than one statement is to be placed on a line, the delimiter `;` can be used.
- If a correct answer requires no action, `if...then ; else...` would be incorrect. A semicolon does not cause a null instruction to be executed; the no-operation instruction `nop` would have to be used, as in `if...then nop else...`.
- When the test fails, the `else` portion is executed. You can include the value of expressions (for example, `a+b`) in the data to be displayed on the console.

For full details on REXX instructions, see the *z/VM: REXX/VM Reference*.

Issuing z/VM Commands

Although the preceding program contains only REXX language statements, a REXX program can also contain CMS and CP commands. The following exec, called RUNCOB1, asks you for the name of the VS COBOL II program you want to compile, compiles the program, and loads and executes the program. (The file type is assumed to be COBOL.) For clarity, the z/VM commands are shown in upper case; the interpreter, however, does not differentiate between uppercase and lowercase (except within strings and literals, for example).

```
/*  RUNCOB1 EXEC  */
Mainpart:
    signal on error
```

```
    say 'What is the name of the file you are compiling?'
    pull a
    'COBOL2' a
    'GLOBAL TXTLIB VSC2LTXT'
    'LOAD' a '(START'
    say 'RUNCOB1 EXEC COMPLETED'
    exit
Error:
    rcsave = rc
    'SET CMSTYPE RT'
    say "Unexpected Return Code" rcsave "from command:"
    say "       " sourceline(sigl)
    say "at line number" sigl "."
    exit
```

The following describes the sequence of execution:

- The return code from commands is placed in the special REXX variable `rc`.
- In the REXX language, a clause consisting of a single symbol followed by a colon is considered a label. The colon acts as an implicit terminator, so no semicolon is required, even when the label is followed on the same line by another statement. (For clarity, the label `Mainpart:` is not indented.)
- The instruction `signal on error` switches on a detector in the interpreter that tests the return code from every command. If a nonzero return code is encountered, the normal sequence of clauses is abandoned and execution transferred to a special label `Error:`. This detector can be switched off by issuing the instruction `signal off error`.
- The REXX control statement `say` prompts the users for the name of the VS COBOL II program to be processed. `Pull` assigns the value, entered as a result of the prompt, to the variable `a`.
- The `COBOL2 a` command compiles the source program.
- The `GLOBAL TXTLIB VSC2LTXT` command identifies the text library VSC2LTXT. CMS searches VSC2LTXT TXTLIB, along with any default system TXTLIBs, for macros, COPY files, subroutines, or modules needed when processing the following LOAD command.
- `LOAD a (START` loads and executes the compiled program. If the `LOAD` command executes successfully, the `RUNCOB1 EXEC COMPLETED` message is displayed.
- When the `signal on error` detector encounters a nonzero return code, the interpreter assigns the line number of the failing command to the special program variable `sigl` and then transfers processing to the label `Error:`. The REXX language function `sourceline (n)` returns the nth line in the source file. Here `sourceline(sigl)` displays the failing line of code. Its position in the display is indented by prefixing it with a literal of six spaces.

EXEC 2 Processor and CMS EXEC Processor

EXEC 2 programs and processing are similar to CMS EXEC programs, with the following differences:

- There is no 8-byte restriction on token length. The words that comprise EXEC 2 statements can be up to 255 characters long.
- You can use EXEC 2 to issue commands to specified subcommand environments, such as the editor macro facility, as well as CMS and CP.
- EXEC 2 has extended string manipulation and arithmetic functions.
- You can define EXEC 2 subroutines and functions.
- EXEC 2 provides extensive debugging facilities.
- CMS user programs can manipulate EXEC 2 variables.

Although CMS EXEC programs can call EXEC 2 programs, and EXEC 2 programs can call CMS EXEC programs, the language statements cannot be mixed within one exec.

Sample EXEC 2 Language Program

The following exec, called ADD, asks you for three numbers that you want to add, determines if you entered numeric data, and adds the three numbers you entered.

ADD EXEC contains the following:

```
&TRACE
&ERROR &EXIT &RETCODE
&TYPE Enter three numbers you want to add:
&READ ARGS
&IF &N = 3 &GOTO -ADDNUMS
&TYPE You must enter three numbers
&READ ARGS
&IF &N = 3 &GOTO -ADDNUMS
&GOTO -ERROR
-ADDNUMS
&TYPE1  = &DATATYPE OF &1
&TYPE2  = &DATATYPE OF &2
&TYPE3  = &DATATYPE OF &3
&IF &TYPE1 NE NUM &GOTO -ERROR
&IF &TYPE2 NE NUM &GOTO -ERROR
&IF &TYPE3 NE NUM &GOTO -ERROR
&SUM = &1 + &2 + &3
&TYPE The sum of  &1  &2  and  &3  is  &SUM
&EXIT
-ERROR
&TYPE You did not enter three valid items.  This program is ending.
&EXIT
```

The following describes the sequence of execution:

- The &TRACE indicates to the system that this exec is written in the EXEC 2 language.
- The &ERROR control statement specifies that if a z/VM command results in a nonzero return code, the &EXIT statement is executed. In this example, the return code, indicated by the control word &RETCODE, is passed upon exit.
- The &TYPE statement asks you to enter three numbers. The &READ ARGS statement reads the items you enter and assigns them to the variables &1, &2, and &3.
- The &IF statement checks to see if you entered three items. If you did not, you are asked again to enter three numbers. If you still do not enter three items, an error message is displayed and the program ends.
- If you entered three items, execution flows to -ADDNUMS. The next six lines check to see if you entered only numbers and not any other characters. The &DATATYPE OF function yields a value of "NUM" if &1, &2, or &3 is a valid number and yields a value of "CHAR" if &1, &2, or &3 is anything else. If &TYPE1, &TYPE2, or &TYPE3 does not contain "NUM", execution flows to -ERROR and the program ends.
- If &TYPE1, &TYPE2, or &TYPE3 contains "NUM", then &1, &2, and &3 are added together and the sum is displayed.

Many EXEC 2 facilities are similar to CMS EXEC facilities. Some control statements and special variables have not been covered here. For full details on the EXEC 2 processor facilities, see the *VM/SP: EXEC 2 Reference*.

Sample CMS EXEC Language Program

The following CMS EXEC program, called RUNCOB2 EXEC, compiles, loads, and executes a VS COBOL II source file:

```
*  RUNCOB2 EXEC  *
&CONTROL OFF NOMSG
&IF &INDEX LT 1 &GOTO -ERR1
COBOL2 &1
&IF &RETCODE NE 0 &EXIT
GLOBAL TXTLIB VSC2LTXT CMSLIB
&IF &RETCODE NE 0 &EXIT
LOAD &1 (START
&IF &RETCODE NE 0 &EXIT
&TYPE RUNCOB2 EXEC FINISHED
&EXIT
-ERR1
&TYPE PROGRAM NAME NOT GIVEN
&EXIT
```

RUNCOB2 EXEC works as follows:

- The control statement &CONTROL sets the type of execution information displayed at the console. No execution messages or return codes are to be displayed here.
- The control statement &INDEX is a special variable that contains the number of arguments entered by the caller. If no arguments are supplied, execution goes to label -ERR1.
- COBOL2 &1 compiles the source program, specified by &1. If the COBOL2 command does not complete successfully, &RETCODE is nonzero and &EXIT is processed causing an immediate exit from the exec. The special variable &RETCODE contains the return code from the most recently executed CMS command. If &RETCODE is zero, the source program compiled successfully.
- If the GLOBAL command fails, &RETCODE is nonzero and &EXIT is processed causing an immediate exit from the exec. If &RETCODE is zero, the TXTLIBs are successfully found and identified.
- LOAD &1 (START loads the compiled program and executes it. If the LOAD command fails, &RETCODE is nonzero and &EXIT is processed causing an immediate exit from the exec. If &RETCODE is zero, the program was loaded and executed and the RUNCOB2 EXEC FINISHED message is displayed.
- If too few arguments are supplied, execution is routed to the label -ERR1, where the warning message is typed. No &EXIT is required here, because processing ends at the end of the exec.

For information on the formats for the CMS EXEC control statements, use the HELP facility by using the HELP command.

Alternate Format Exec

An alternate format exec is executed using a processor other than the REXX/VM interpreter, the EXEC 2 processor, or CMS EXEC processor. Alternate format execs can be written in any language, including one that you create. The first line (the header record) distinguishes an alternate format exec from other exec languages. The header record format is described later.

An alternate exec processor is a language processor that you write or is provided to you through your particular installation. When CMS determines that the current command is an alternate format exec, it invokes the alternate exec processor and passes information about the exec in the extended parameter list and in registers 0 and 1.

Once the alternate exec processor receives an exec and register information, it can execute the alternate format exec by doing one or more of the following: interpreting the exec, enforcing its language syntax rules, or performing any other required processing.

Naming Conventions for Alternate Format Execs

Alternate format execs must follow the CMS file naming conventions if they are to be recognized as valid exec programs.

In addition, if you are developing an alternate format exec using the REXX language as source, you may want to use the REXX/VM interpreter as a debugging tool. In this case, the file name of the exec you use as input to the interpreter be the same as the file name you use as input to the alternate exec processor.

Header Record Format of Alternate Format Execs

The header record distinguishes an alternate format exec from execs written in the REXX, EXEC 2, or CMS EXEC language. The format of this record is:

Offset	Length	Description
+0	4	Any 4 bytes where the first two characters are not "/*" and the first character is not “*”.
+4	8	The string 'EXECPROC' (must be uppercase).
+12	8	The name of an alternate exec processor, left-justified, padded with blanks if necessary, and in conformance with all the rules for CMS command names.

The remaining portion of an alternate format exec is in the exec language or in a format agreed upon by the alternate format exec and its processor.

Calling the Alternate Exec Processor

Once an alternate format exec is recognized as a valid exec, some of the ways you can initiate them are:

- From the command line
- From another exec
- As an XEDIT macro
- In a REXX clause
- From a module.

Alternate format execs can also be:

- Loaded into storage with the EXECLOAD command
- Queried with the EXECMAP and EXECSTAT commands
- Dropped from storage with the EXECDROP command
- Loaded into a segment with the DCSSGEN or SEGMENT LOAD command.

Register Contents and File Status

Upon recognition of an alternate format exec, the CMS EXEC interface issues a CMSCALL macro (SVC 204) for the alternate exec processor named in the header record. On entry, the alternate format exec can expect:

- R0 pointing to an extended plist that has the same format as one of the extended plists supported by REXX. The fourth word always contains the address of an FBLOCK. An alternate exec processor can extract any information about an exec from the extended plist. See the *z/VM: REXX/VM Reference* for more information.
- R1 pointing to a tokenized plist consisting of an 8-byte long string containing the name of the alternate exec processor followed by fields containing:
  - The values of R2, R1, and R0 on entry to the CMS EXEC interface.
  - The address of an FBLOCK describing the exec to be processed. The FBLOCK contains only the file name, file type, file mode, descriptor list address, and descriptor list length. If the exec is in storage, the file mode is "*".
  - An 8-byte fence, X'FFFFFFFF'.

  Note: The value of R2 on entry to the CMS EXEC interface is provided to allow the alternate exec processor the ability to implement the non-SVC subcommand interface. This interface is documented in the *z/VM: REXX/VM Reference*.
- All other registers set according to the CMSCALL process. See the *z/VM: CMS Application Development Guide for Assembler* for more information.
- The file containing the alternate format exec to be in OPEN state. This file remains OPEN until end-of-command unless explicitly closed by the alternate exec processor.

CMS Services Available to the Alternate Exec Processor

In addition to the standard services provided by CMS, alternate exec processors have access to the following CMS services:

- Terminal input buffer line count. See the *z/VM: REXX/VM User's Guide* for more information.
- Disk block size determination. You can obtain this information using the Extract/Replace facility. See the *z/VM: CMS Callable Services Reference* for more information.
- The non-SVC subcommand invocation that uses a minimum overhead subcommand call for issuing commands. See the *z/VM: REXX/VM Reference* for more information.

- The CMS global exit facility and the REXEXIT macroinstruction. See the *z/VM: CMS Macros and Functions Reference* for more information.
- Recursion control when using the CMS subcommand interface.
  - If you want to inhibit the recursion of execs, the fourth word of the extended plist must be nonzero. It must be the address of a CSFCB containing a pointer to the 8-byte name of the current exec.
  - If you do not want to inhibit the recursion of exec, the fourth word of the extended plist must be zero (that is, there is no CSFCB).

  See the *z/VM: REXX/VM Reference* for information on CMS command search order and how it effects calling execs recursively.
- CMS ensures that on entry to any alternate exec processor, both tokenized and extended plists are available. When generating calls to other commands or execs, an alternate exec processor must provide a tokenized plist and optionally provide an extended plist. It must also set the user call-type information to indicate the format of the plists provided. See the *z/VM: CMS Application Development Guide for Assembler* for more information about creating and passing plists.

  If only a tokenized plist is available, the CMS EXEC interface generates an extended plist from information in the tokenized plist, before invoking the language processor of the called exec.

  Note: Although an alternate exec processor can execute above the 16 MB line, when generating calls to an exec written in EXEC 2 or CMS EXEC language, an alternate exec processor must ensure that the parameter lists and the data referenced by the parameter lists reside below the 16 MB line.

Creating an XEDIT Macro

You can write execs to be used with XEDIT. These execs are known as XEDIT macros. You use them when you use the editor to create or edit a file. These execs have a file type of XEDIT rather than EXEC. You must use the REXX language or the EXEC 2 language when you write XEDIT macros. Otherwise, they are like ordinary execs.

For example, the following macro places continuation characters on specified lines in the correct column of COBOL or FORTRAN files. (For clarity, XEDIT commands are in upper case.)

```
/* CONTCHAR XEDIT */ Mainpart:
    'SET MSGMODE OFF'
    'PRESERVE'
    numlines = 1
    if arg() > 0 then numlines = arg(1)
    'EXTRACT/FTYPE/'
    col = 72
    if ftype.1 = 'COBOL'   then col = 7
    if ftype.1 = 'FORTRAN' then col = 6
    'SET TRUNC' col
    'SET ZONE' col col
    'CHANGE/ /*/' numlines
    'RESTORE'
    'SET MSGMODE ON'
    exit
```

This is what the macro does:

- The message display option (SET MSGMODE) of the Editor is set off when you use the macro. Thus, its execution appears like a regular XEDIT subcommand.
- The PRESERVE command ensures that the settings of the various XEDIT variables, such as line length, are retained until the RESTORE command is executed.
- If an argument is supplied, the variable `numlines` is set to its value. Otherwise, it remains equal to 1.
- The XEDIT subcommand EXTRACT makes the file type of the file being edited available in the variable `ftype.1`.
- The variable `col` is set to 6, 7, or 72. This value defines the truncation column and then to set the zone.

- The CHANGE subcommand causes continuation characters to be included. The macro ends by resetting the environment to its original state.

PROFILE EXEC File

You can create a special exec file, called PROFILE EXEC, to link to a disk and define its access order, set up some characteristics for the terminal, and initialize some macro libraries.

```
CP LINK DEWEY 193 193 RR
ACC 193 B/A
CP SET EMSG ON
CP TERM HILIGHT ON
GLOBAL MACLIB DMSGPI OSMACRO
```

Such commands are typically issued at the start of every terminal session. The PROFILE EXEC is automatically executed the first time you press Enter after CMS is loaded.

For more information on using a PROFILE EXEC, see the *z/VM: CMS User's Guide*.

CMS EXEC File

You can create a special EXEC file, called CMS EXEC, by using the LISTFILE command with the EXEC option.

Suppose you have a series of files on your disk with file names beginning with the characters "PAY" and file types beginning with the character "D". If you enter:

```
listfile pay* d* a (exec
```

the usual LISTFILE display is placed in a file CMS EXEC. It has the format:

```
&1 &2 filename filetype filemode
```

Assume that after you entered the LISTFILE command shown, the CMS EXEC file contains:

```
&1 &2 PAYROLL   DATA       A
&1 &2 PAYDATE   DOCUMENT   A
&1 &2 PAYSLIP   DETAIL     A
&1 &2 PAY23UPD  D831102    A
```

If you now enter:

```
cms disk dump
```

CMS executes the following commands:

```
DISK DUMP PAYROLL DATA A
DISK DUMP PAYDATE DOCUMENT A
DISK DUMP PAYSLIP DETAIL A
DISK DUMP PAY23UPD D831102 A
```

The arguments DISK and DUMP replace &1 and &2 when the file is executed.

If only one argument is passed to an exec, the succeeding variables are set to nulls. For example, if you enter:

```
cms erase
```

CMS executes the following commands:

```
ERASE PAYROLL DATA A
ERASE PAYDATE DOCUMENT A
ERASE PAYSLIP DETAIL A
ERASE PAY23UPD D831102 A
```

The CMS EXEC file is like any other CMS file. You can edit it, print it, sort it, and rename it. Each time you use LISTFILE with the EXEC option, a new CMS EXEC is created and the old one is erased. You can add to an existing CMS exec file using LISTFILE with the APPEND option.

Using the FILEDEF Command in Execs

You can use the FILEDEF command to identify to z/VM the I/O files of an OS program. The FILEDEF command can be used in execs just like other z/VM commands, and can eliminate multiple lines of typing before a program is executed.

The following example demonstrates this by using the FILEDEF command inside a loop. This is possible because the ddnames and file types each contain a unique number as the last character.

```
/* set up payroll files */
Mainpart:
     say "Payroll Files - Weekly or Monthly (W/M)?"
     pull runtype
     say "How many Overtime files?"
     pull otime
     signal on error
     select
       when runtype = 'W' then
         do
         'FILEDEF INFILA C1 DSN STAFF.WEEKLY.PAYFILEA'
         'FILEDEF INFILB C1 DSN STAFF.WEEKLY.PAYFILEB'
         end
       when runtype = 'M' then
         do
         'FILEDEF INFILA C1 DSN STAFF.MONTHLY.PAYFILEA'
         'FILEDEF INFILB C1 DSN STAFF.MONTHLY.PAYFILEB'
         end
       otherwise
         do
         say "Incorrect reply - must be W or M - please restart"
         exit
         end
     end
     do while otime > 0
         'FILEDEF OTFIL'||otime 'DISK OVERTIME DATA'||otime 'B4'
         otime = otime - 1
         end
     'FILEDEF MASINP DISK STAFF MASTER1 B4'
     'FILEDEF MASOUT DISK STAFF MASTER2 A4'
     'FILEDEF CONSOL TERM'
     exit
Error:
     say "Failure to execute FILEDEF command at"
     say "line number" sigl "Return code" rc
```

This exec first requests the type of payroll file (PAYFILE) and then the number of overtime files to be processed. Depending on the type, either weekly or monthly data sets are identified on the OS disk (in this example, the C disk).

The exec then uses FILEDEF to relate the internal ddnames of the form OTFIL*n* (where *n* is a number) to the overtime files. These are in OS simulated data set format (on the B disk) and have CMS identifiers OVERTIME DATA*n* B4 (where *n* is the same number used in the ddname).

The next three FILEDEF commands identify the master input and output files with ddnames MASINP and MASOUT respectively, and the z/VM terminal with the ddname CONSOL.

Using MACLIBs and TXTLIBs in Execs

You can write an exec that contains all the front-end commands needed to execute your applications, such as the FILEDEF and GLOBAL commands. This saves you from issuing separate commands every time you compile, load, and execute your applications.

The following is a partial exec containing FILEDEF and GLOBAL commands:

```
/* compile a cobol prog */
Mainpart:
   signal on error
        .
        .
        .
   arg progname privlib
        .
        .
        .
   'FILEDEF' ......
        .
        .
        .
   'GLOBAL MACLIB' privlib 'OSMACRO'
        .
        .
        .
   'COBOL2' progname 'COBOL'
        .
        .
        .
   say 'Any extra TXTLIBs required?'
   pull txtlibr
        .
        .
        .
   if arg () = 0 then
       'GLOBAL TXTLIB VSC2LTXT'
     else
       'GLOBAL TXTLIB' txtlibr 'VSC2LTXT'
        .
        .
        .
   exit
ERROR:
        .
        .
        .
```

- This exec processes the compilation, link editing, and execution of a COBOL program. First you must supply two parameters: the program name (`progname`) and the programmer's private MACLIB (`privlib`). The GLOBAL command parameters are ordered so that the compiler searches the private library before the standard OS and TSO libraries.
- `signal on error` ensures that a nonzero return code from the call to COBOL2 causes execution to be routed to the label ERROR:. Otherwise, you must request any additional TXTLIBs.
- The GLOBAL command handles text libraries in a similar fashion to macro libraries. If you specify a private TXTLIB, `txtlibr`, GLOBAL incorporates it with the installation's library, VSC2LTXT.

Prototyping with REXX

REXX makes it easy for you to prototype algorithms before they are included in a larger compiled program. This procedure leads to faster program development, since design bugs are more quickly trapped without the need for multiple compilations. As mentioned in “Restructured Extended Executor Language” on page 333, although the interpreter does not execute as efficiently as a compiler, it takes less time to develop a program. Therefore, sizable savings result.

Here is an example:

```
/* Square Root Exec */; arg val;tol = 0.0001; old=0; new=1; count=0;
do while abs(old - new) > tol
       old = new
       work1 = old ** 2 + val
       work2 = 2 * old
       new = work1/work2
       say new
       count = count + 1
       end; say 'RESULT =' new 'CYCLES =' count; exit
```

This routine tests an algorithm for calculating square roots. Before incorporating it into a final compiled program, you can test its accuracy using a REXX procedure.

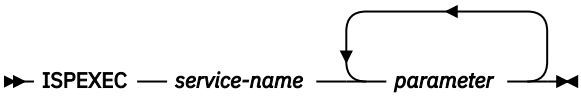
The procedure accepts the value whose root is required as an argument. As the main loop is executed, the current approximation to the root is displayed. At the end, the result is displayed, together with the number of cycles required to calculate it.

When you use REXX as a prototyping tool, you should be careful when calling functions and subroutines. The mechanism the interpreter uses to pass arguments and results may not correspond to the mechanism of the compiled language that will eventually be used.

Prototyping Interactive Applications

You can conveniently use execs with ISPF applications before implementing the application in a high-level language, such as COBOL or FORTRAN. The exec can invoke sequences of DISPLAY and SELECT services and handle related variables.

EXEC 2 and REXX give you the facility to call ISPF panel and variable services. In EXEC 2, the format of a call to an ISPF service is as follows:



Variables can be used anywhere in the statement as the service name or as a parameter. Each variable is replaced with its current value before execution of the ISPEXEC command. You can use parameter keywords wherever they apply. Otherwise, the parameters are positional. Here are some EXEC 2 examples:

```
&SUBCOMMAND ISPEXEC DISPLAY PANEL(&PNAME )
&SUBCOMMAND ISPEXEC DISPLAY PANEL(MENUPAN)
```

In the first example, the EXEC 2 variable &PNAME is passed as a parameter. It is assumed to have a value MENUPAN. In the second example, this value is passed directly. You do not have to incorporate the keyword PANEL, because the parameter is in the first position. EXEC 2 requires that you precede ISPEXEC with a &SUBCOMMAND unless the statement:

```
&PRESUME &SUBCOMMAND ISPEXEC
```

is included in the procedure before executing the first ISPEXEC command. Some ISPF services allow dialog variables names to be passed as parameters. If you pass such names in EXEC 2, do not precede them with an ampersand. For example:

```
ISPEXEC VGET XYZ
```

Here XYZ is the name of the dialog variable to be passed. The VGET service can also accept a list of variables passed as a single parameter. If you pass such a list, you must enclose it in parentheses. You must also separate the items with blanks or commas. For example:

```
ISPEXEC VGET (AAA,BBB,CCC)
ISPEXEC VGET (XXX YYY ZZZ)
```

Here are some REXX examples:

```
Address ISPEXEC DISPLAY PANEL '('PNAME')'
Address ISPEXEC DISPLAY PANEL '(menupan)'
```

In the first example, PNAME is a REXX variable assumed to have a value MENUPAN. In the second example, the value is passed directly.

ISPEXEC operations end with a return code in the same way as other routines do. Thus, you can use &RETCODE or &RC in EXEC 2 or RC in REXX to test the success of the calls.

Here is an example of using ISPEXEC in an EXEC 2 procedure:

```
&TRACE OFF
&PRESUME &SUBCOMMAND ISPEXEC
CONTROL ERRORS RETURN
TBOPEN EMPLTBL
&IF &RC EQ 0 &GOTO -CONT1
TBCREATE EMPLTBL (EMPSER) (LNAME FNAME)
-CONT1
&F =
&EMPSER =
VPUT (F EMPSER)
DISPLAY PANEL (MENUPAN)
&IF &RC EQ 8 &GOTO -EXIT
&IF &F GT 4 &GOTO -EXIT
TBGET EMPLTBL
&IF &F EQ 1 &IF &RC NE 0 &GOTO -CONT2
&IF &F GT 1 &IF &RC EQ 0 &GOTO -CONT3
SETMSG MSG (MSG002 )
&GOTO -CONT1
-CONT2
&FNAME =
&LNAME =
-CONT3
SETMSG MSG (MSG001 )
&IF &F EQ 3 &GOTO -CONT4
DISPLAY PANEL (NAMEPAN)
&IF &F EQ 1 &GOTO -CONT5
&IF &F EQ 2 &GOTO -CONT6
&GOTO -CONT1
-CONT4
TBDELETE EMPLTBL
&GOTO -CONT1
-CONT5
TBADD EMPLTBL
&GOTO -CONT1
-CONT6
TBPUT EMPLTBL
&GOTO -CONT1
-EXIT
TBCLOSE EMPLTBL
&EXIT
```

This exec invokes a number of ISPF functions.

- CONTROL ERRORS RETURN tells ISPF to return to dialog processing when an error condition occurs (instead of terminating).
- TBOPEN tells ISPF to open the table EMPLTBL, if it exists. If the table does not exist, a nonzero return code is issued. The exec tests the return code, and if it is nonzero, the EMPLTBL table is created using the TBCREATE function.
- VPUT tells ISPF to initialize the panel variables specified (in this case, variables F and EMPSER). They are initialized to blanks before the panel display.
- DISPLAY tells ISPF to display the specified panel (MENUPAN or NAMEPAN) on the screen.
- TBGET tells ISPF to retrieve values from the EMPLTBL table.
- SETMSG tells ISPF to display the specified message on the next panel.
- TBDELETE tells ISPF to delete the current record (the one specified on the panel) from the EMPLTBL table.
- TBADD tells ISPF to add the current record to the EMPLTBL table.
- TBPUT tells ISPF to update the current record in the EMPLTBL table.
- TBCLOSE tells ISPF to close the EMPLTBL table.

If you want to run this exec, you have to do the following:

1. Issue various FILEDEF commands to specify the ISPF libraries to be used. The necessary FILEDEFs to run the exec are listed in Appendix J, “ISPF Example,” on page 599.

2. Create a panel library called USERPAN MACLIB containing the MENUPAN COPY and NAMEPAN COPY files.
3. Create a message library, EXAMMSG MACLIB, containing definitions of messages MSG001 and MSG002. All panels and messages are described in Appendix J, “ISPF Example,” on page 599.
4. Invoke ISPF with the CMD parameter, instead of PGM when using an exec.

For more details on the use of execs with ISPF, see *ISPF Dialog Management Guide and Reference*.

Using Execs with ISQL

You can easily use ISQL (Interactive SQL) when you access or update portions of tables occasionally. When you access or update tables more frequently, you may want to group sets of ISQL commands into an exec. An efficient way to execute these commands is to collect all the data you need, call ISQL, do the necessary commands, and return to z/VM.

z/VM provides a stack for commands to be executed on a first-in first-out basis. You can place items onto the stack using the REXX command QUEUE.

Here's an example of an exec to change an entry in a phone list:

```
/* changnum exec  */
   say 'Supply last name'
   pull lnam
   say 'And now the initial'
   pull init
   say 'Enter new phone number'
   pull nnum
   queue 'COMMIT WORK'
   queue 'UPDATE PHONELIST -'
   queue 'SET PHNUM = 'nnum' -'
   queue 'WHERE LNAME = 'lnam' AND FINTL = 'init
   queue 'COMMIT WORK'
   queue 'EXIT'
   exec  'ISQL'
```

In this example:

- When the necessary details are supplied to the exec, the sequence of ISQL commands are placed onto the command stack, using the QUEUE command.
- The exec variables set to the supplied details are contained within the ISQL commands inside an inner set of quotation marks. (For clarity, the example shows them in lower case.)
- When the list is updated, the work is committed. Because we want to return to z/VM after the operation, the ISQL command EXIT is stacked.
- Finally, the exec invokes ISQL to start processing the commands queued on the stack.

Because the interpreter executes in the CMS environment, it is not available while you are running ISQL. The stack gives you a way to transfer commands from one to the other. In this case, you begin and end in CMS.

You can also build execs you expect to execute during ISQL sessions. Inside the exec you set a RETURN command as the first thing in the stack. You no longer need the QUEUE EXIT and EXEC ISQL commands at the end of the exec. To start the exec during an ISQL session, you can enter "CMS" to get back into the CMS mode. You can then enter "CHANGNUM" (the name of the exec.)

When you supply the data and the exec ends, the RETURN command in the stack is executed. This takes you from CMS back into the ISQL environment. The rest of the stacked items (ISQL commands) are then processed. Because there is no EXIT at the end of the stack, you remain in the ISQL environment. Besides using this type of exec for less complex SQL table operations, you can use it for prototyping database operations during design and development stages.

There is a direct interface into DB2 Server for VM called RXSQL. For more information on RXSQL see the *RXSQL Reference*.

Chapter 21. Passing Commands and Data

This chapter describes:

- What a stack is and how it is used.
- How to manipulate a program stack.

Stacks

In CMS, the *console stack* is used for input or output. The console stack consists of the *program stack* and the *terminal input buffer*. The program stack is used to pass data to certain CMS commands, or to obtain data from them. It can be used both as a stack (LIFO – Last In/First Out) or a queue (FIFO – First In/First Out). You can build extensions to the program stack which are called buffers. User applications or execs control the number of buffers and the data that reside within each buffer. The terminal input buffer stores data from the CMS command line when you type ahead and press enter while a previous command is still executing. Under certain situations, CMS uses the terminal input buffer internally and transparently. Data placed on the terminal input buffer is always processed on a FIFO basis.

When CMS reads a command, or when a REXX exec issues the PULL instruction to read data, all levels of the program stack are checked first for an entry, followed by the terminal input buffer. If there is nothing on the program stack or terminal input buffer, then execution is suspended and 'VM READ' or 'Enter your response in vscreen VNAME' will be displayed in the status area. Execution will not resume until data is entered at the console.

Figure 49 on page 347 shows the structure of the *console stack*. For more information, see the *z/VM: REXX/VM User's Guide*.

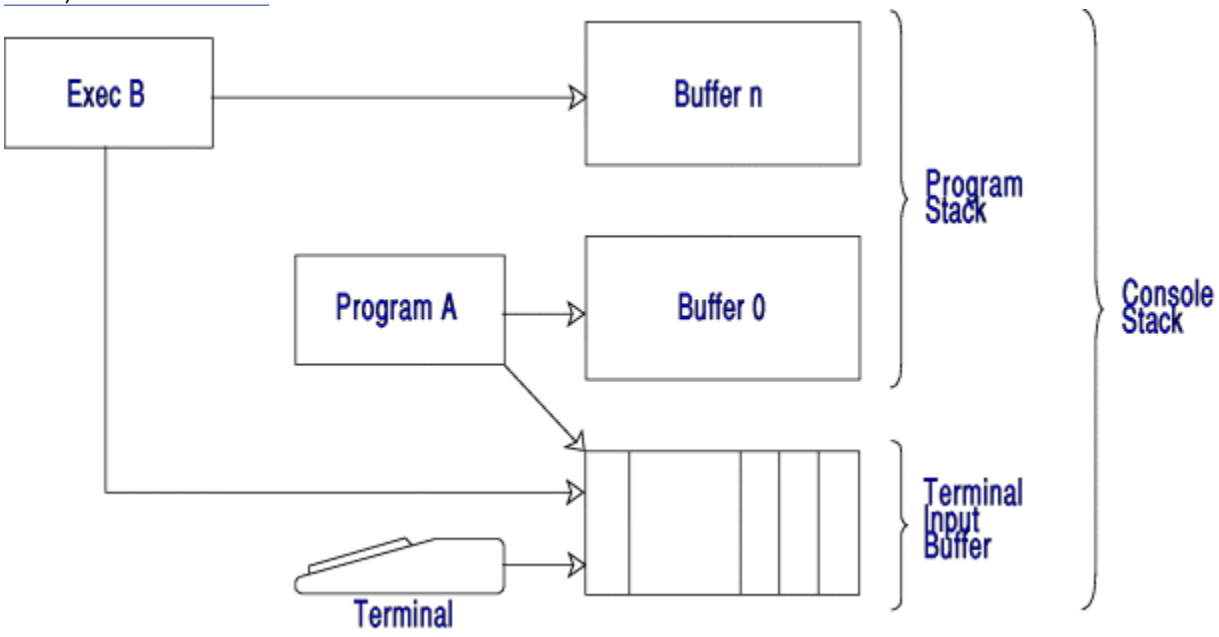


Figure 49. Elements of a Console Stack

Using the Program Stack to Pass Data Between Programs

The following example describes one way of how you can use the program stack. CMSHELLO EXEC first asks for your first name, last name, and the language of your source program. Then CMSHELLO EXEC places your first name and last name on the program stack. Then, depending on the language of your source program, CMSHELLO EXEC places the name of the source program on the program stack and calls the corresponding compiler exec to compile, load, and execute a source program.

Suppose, in our example, the source program is a FORTRAN program. Therefore, CMSHELLO EXEC identifies HELLO FORTRAN as the source program and calls the compiler exec called COMPFORT EXEC. COMPFORT EXEC compiles, loads, and executes the HELLO FORTRAN program. The HELLO FORTRAN program welcomes you to CMS by displaying a message.

You do not have to respond to program prompts in the source program because the HELLO FORTRAN reads the lines stacked in CMSHELLO EXEC. By using CMSHELLO EXEC, you do not have to know the name of the source program that welcomes you to CMS or the commands needed to compile, load, and execute the source program.

CMSHELLO contains the following:

```
/* This is a setup exec to determine what source file
   will be used to welcome you to CMS.                */
say 'What is your first name?'
pull firstname
say 'What is your last name?'
pull lastname
say 'What language are you using:  FORTRAN, COBOL, PL/I, or C?'
pull lang
/* Putting your last name and first name on the stack. */
push lastname
push firstname
/* Determining what language you are using, placing the name of the
   corresponding source file on the stack, and invoking the
   correct compiler exec.                                          */
select
  when lang='FORTRAN' then do
    push 'HELLO'
    'EXEC COMPFORT'
    end
  when lang='COBOL' then do
    push 'HELLO'
    'EXEC COMPCOB'
    end
  when lang='PL/I' then do
    push 'HELLO'
    'EXEC COMPPLI'
    end
  when lang='C' then do
    push 'HELLO'
    'EXEC COMPC'
    end
  otherwise
    say 'No language was specified.  Your program has ended!'
end
'DROPBUF'
exit
```

COMPFORT EXEC contains the following:

```
/* the executor language version of COMPILE EXEC */
'SET CMSTYPE HT'
Mainpart:
    signal on error
    pull pname
    'FORTVS2' pname
    if rc¬=0
      then signal on error
      else do
        'SET CMSTYPE RT'
        'GLOBAL TXTLIB VSF2FORT CMSLIB'
        'GLOBAL LOADLIB VSF2LOAD'
        'LOAD' pname
        'GENMOD' pname
        pname
        end
    exit
    Error:
    rcsave = rc
    'SET CMSTYPE RT'
    say "Unexpected return code" rcsave "from command at line number "sigl":"
    say "     " sourceline(sigl)
    exit
```

To invoke this exec, enter:

```
cmshello
```

Once you answer the prompts, the following appears on the screen:

```
What is you first name?
John
What is your last name?
Doe
What language are you using: FORTRAN, COBOL, PL/I, or C?
FORTRAN
Welcome to CMS, John     Doe
```

Using the Program Stack to Pass Data to CMS

Another way the program stack can be used is as a mechanism to pass data between your program and CMS. SORTPRT EXEC in Figure 50 on page 349 demonstrates this interaction. Assume you want to sort a file in alphabetic order before it is printed, but you do not want to change the original file. SORTPRT EXEC in Figure 50 on page 349 does this by using the CMS SORT command. The SORT command prompts you for the sort fields and will create a new file containing the sorted records. But, because you have queued the response to the prompt on the program stack, you will not have to enter in the sort fields at the console. Remember, when the read for the prompt is done, the program stack is checked first for entries.

You will notice that SORTPRT EXEC also creates its own buffer on the program stack by issuing the CMS MAKEBUF command. Before exiting, it deletes this buffer by issuing the CMS DROPBUF command. This is to insure that any other stack activity is not affected if the SORT command encounters an error before reading the data from the program stack.

```
/* Sort and print */
'MAKEBUF'
QUEUE "1 8"
'SORT DATA FILE A WORKDATA TEMP A'
if rc¬=0 then do
      say "unexpected return code",
      rc "from sort command"
      end
else 'PRINT WORKDATA TEMP A'
'DROPBUF'
exit
```

Figure 50. SORTPRT EXEC

Figure 51 on page 350 illustrates how local buffers affect the system when you do the local stack example. The following takes place:

1. Your program gets control.
2. Your program stacks the message for the CMS sort.
3. Your program issues the SORT command, transferring control.
4. The SORT command reads from the console stack and performs the sort.
5. Your program performs the final steps, then exits.
6. CMS regains control.

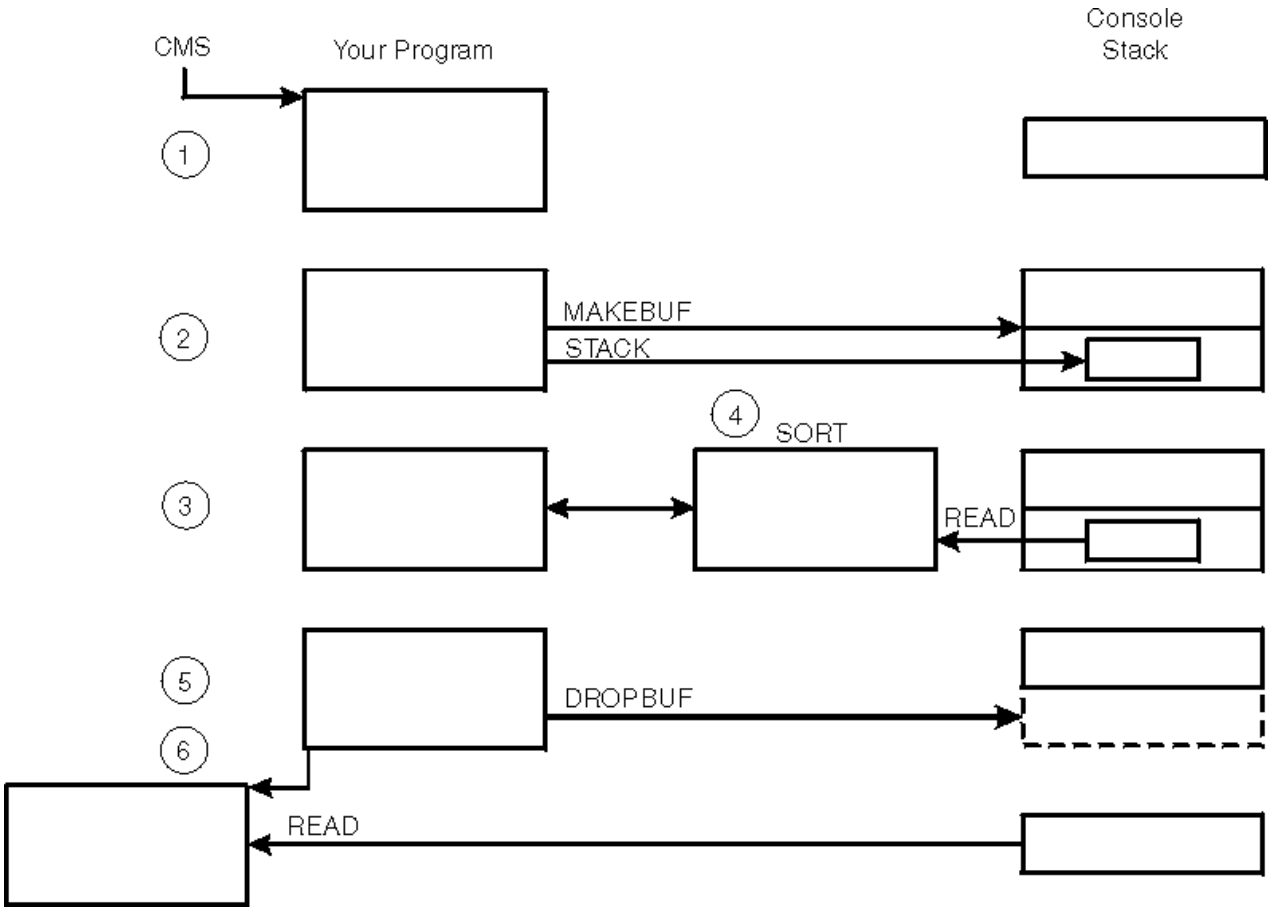


Figure 51. Example of Local Stack Usage

Manipulating the Program Stack

Some statements and commands that put data into a program stack are:

- For the REXX language:

  QUEUE
  queues data FIFO.

  PUSH
  pushes data LIFO.

  CHAROUT
  writes characters to the stack

  LINEOUT
  writes lines to the stack

- For the CMS EXEC or EXEC 2 language:

  &STACK FIFO
  queues data FIFO.

  &STACK LIFO
  pushes data LIFO.

Some statements and commands that read data from a program stack are:

- For the REXX language:

  PULL
  reads the next item.

CHARIN
reads characters from the stack

LINEIN
reads lines from the stack

- For the CMS EXEC or EXEC 2 language:

&READ
reads the next item.

- Any CMS commands that access stack elements.

The order in which stacked items are retrieved is determined at the time they are placed in the stack.

CMS commands, macros, and callable service library routines that manipulate the program stack are listed in "Program Stack I/O" on page 115. They can be used in assembler and many high-level languages.

Using Program Stacks

The best way to use the program stack is to maintain control of it with the MAKEBUF and DROPBUF commands. The MAKEBUF command creates a new buffer within the program stack. The buffer number for the new buffer is returned in register 15 by CMS.

Note: If an &ERROR statement is in effect in an EXEC that invokes this command, the return code causes it to execute. Therefore, it is important to ensure that no &ERROR statement is in effect at the time.

After the MAKEBUF command is issued, you can determine how many entries are already on the program stack by issuing the REXX QUEUED() function. This instruction returns the number of entries in the stack (A similar function is available with the SENTRIES command.). Here again, the value is returned in register 15. Be careful not to create an incorrect execution of the SIGNAL ON ERROR statement. The result of the QUEUED function or SENTRIES command can be used by the program as a processing cutoff to avoid using any stack elements from another program or the terminal input buffer.

Chapter 22. Using CMS Pipelines

CMS Pipelines introduces to CMS a data flow model approach to programming. CMS Pipelines lets you solve a large problem by breaking it up into a series of several smaller, less complex programs. These program, called stages, then can be hooked together to form a pipeline to give you the results you want.

Some advantages of using CMS Pipelines are:

- Numerous CMS Pipelines stage commands and pipeline subcommands provide the ability to transform data in many different ways.
- CMS Pipelines provides an interface that allows you to write your own stage commands in the REXX or Assembler language.
- Code is reused each time a stage is run in a pipeline and requires no modification or recompilation.
- Using CMS Pipelines simplifies application development because there are no device interface dependencies.

You must use the PIPE command to invoke CMS Pipelines. You can use the PIPE command from the command line or from an application. This chapter discusses using CMS Pipelines from REXX, EXEC 2, or Assembler programs. It includes:

- Concepts and functions of CMS Pipelines
- Writing execs using CMS Pipelines
- Writing your own stage commands.

For complete information on CMS Pipelines, refer to the *z/VM: CMS Pipelines User's Guide and Reference*.

Basic Concepts and Functions of CMS Pipelines

The operand of the PIPE command consists of at least one **pipeline**. A pipeline is a series of stages, separated by a character called a **stage separator**.

A stage reads data, processes the data, and writes the data to the next stage. The output from one stage is the input to the next stage. Each stage reads its input and writes its output through a device-independent interface without concern for other stages in the pipeline. Because data enters a pipeline and moves through a pipeline's stages, the data is said to flow through a pipeline. The flow of data records into a stage is called an **input stream** and the flow of data out of a stage is called an **output stream**.

CMS Pipelines includes an extensive set of stage commands that:

- Issue CP and CMS commands
- Read from or write to your terminal
- Read, create, or append to a CMS file
- Read a CMS file backwards
- Copy records
- Select a subset of records
- Manipulate multiple records
- Rearrange the contents of records
- Sort records
- Discard or retain duplicate records
- Truncate records
- Count lines, blank-delimited words, and bytes
- Block data

- Deblock external data formats.

Filters, device drivers, and host command interfaces are three types of stage commands. Filters transform data by performing operations such as sorting, truncating, and selecting records. Device drivers and host command interfaces read data from or write data to a device, CMS, CP, or other host environments (such as, DB2, ISPF, XEDIT). Another term for these types of stage commands is built-in, meaning their functions are provided by CMS Pipelines. A user-written stage command may be built with pipelines subcommands and Assembler macros when there is no built-in function available to perform the required task. Both built-in and user-written stage commands are invoked with the PIPE command.

To further extend the versatility of CMS Pipelines, an unlimited number of interconnected pipelines are supported. One PIPE command can specify multiple pipelines.

Using CMS Pipelines in Execs

When first using CMS Pipelines in execs, you may just put a commonly used PIPE command in an exec so you will not have to type the whole PIPE command each time.

You can also use CMS Pipelines to enhance the functions already provided by REXX/VM. For example, CMS Pipelines provides a set of stage commands that work with REXX variables. Using the STEM stage command and the SORT stage command, you can use the following PIPE command in an exec to sort in ascending order the items in the stemmed array `bananas` and put them in the stemmed array `bunch`.

```
/*   */
⋮
"pipe stem bananas. | sort | stem bunch."
⋮
exit
```

You can also use CMS Pipelines to convert data in REXX variables. For example, the C2F operand of the SPECS stage command converts System/390®® internal format for double-precision floating-point numbers to scientific notation. CMS Pipelines provides many other conversion operands on the SPECS stage command.

Using the function CMS Pipelines provides, you no longer need to use the EXECIO command. CMS makes reading from and writing to a file, printer, punch, reader, tape, stack, and variables much easier. See the *z/VM: CMS Pipelines User's Guide and Reference* for information on using stage commands rather than the EXECIO command.

Using REXX and CMS Pipelines, you can specify arguments on a user-written stage command and your REXX program can receive them as arguments using the PARSE ARG instruction.

Calling CMS Pipelines from Assembler Programs

You may have an existing Assembler program that can just be modified to call a pipeline, perform a function, and return to continue processing. The CMSCALL in your program will actually pass control to CMS Pipelines, and return control to the Assembler program when the pipeline is finished processing.

An example showing the CMSCALL to a pipeline follows:

```
ASMPIPE  CSECT
ASMPIPE  AMODE ANY
ASMPIPE  RMODE ANY
         STM   R14,R12,12(R13)
         USING ASMPIPE,R12
         LR    R12,R15
         ST    R13,SAVEAREA+4
         LA    R15,SAVEAREA
         ST    R15,8(R13)
         LR    R13,R15
         CMSCALL CALLTYP=CMS,PLIST=PC,EPLIST=EPLIST
         L     R13,SAVEAREA+4
         LM    R14,R12,12(R13)
         SLR   R15,R15
         BR    R14
         DS    0D
PC       DC    CL8'PIPE'
         DC    CL8'LITERAL'
         DC    CL8'RUN A'
         DC    CL8'SIMPLE'
         DC    CL8'PIPELINE'
         DC    CL8'|CONS'
PIPE     DC    C'PIPE'
PARAMS   DC    C'literal Run a simple pipeline|cons'
DONE     EQU   *
EPLIST   DC    A(PIPE)
         DC    A(PARAMS)
         DC    A(DONE)
SAVEAREA DS 18F
         REGEQU
         END
```

Figure 52. Assembler Program to Run a Pipeline

Use the High Level Assember Language compiler (HLASM) to compile your program, then load it into a module and run it by issuing:

```
run asmpipe
```

Programming Tips When Using CMS Pipelines

When your PIPE command contains many stages, putting each stage on a separate line makes it easier to write and read a PIPE command. For example:

```
/* SAMPLE EXEC */
'pipe',
   'stage1',
   '| stage2',
   '| stage3',
   '| stage4'
exit rc
```

Figure 53. Format for Writing a PIPE Command

Writing Your Own Stage Commands

When CMS Pipelines does not provide the necessary stage commands to perform a particular function, you can write your own stage commands. These user-written stage commands are REXX exec procedures. The file type of a user-written stage command should be REXX. The significant difference between user-written stage commands and other REXX programs is that user-written stage commands also interact with CMS Pipelines. The PIPE command sets up its own subcommand environment and pipeline subcommands in the user-written stage command are recognized by CMS Pipelines.

You use user-written stage commands in a pipeline the same way you use the built-in stage commands provided by CMS Pipelines; as operands on the PIPE command. A pipeline that consists of several stages can be cumbersome and time-consuming to retype each time at your terminal. In cases like this, it is more efficient to put this sequence of stages in a user-written stage command where it can be modified and reused. The sequence of stages is written as the operand of the CALLPIPE pipeline subcommand,

all within the user-written stage command. Basically, you are inserting a new section of a pipeline in the pipeline that called your user-written stage command.

Unlike the CALLPIPE pipeline subcommand which runs another pipeline of many stage commands, you can write a stage command using subcommands to perform the function of a single stage command. For instance, you can use a combination of the READTO and OUTPUT pipeline subcommands to read one record from an input stream and write the record to an output stream. To accompany the READTO and OUTPUT pipeline subcommands, CMS Pipelines provides a set of pipeline subcommands that you can use to manipulate and modify the records that are read by READTO and written by OUTPUT.

Figure 54 on page 356 shows the basic programming model of a REXX user-written stage command.

```
/* MYSTAG REXX */
signal on error
do forever
   'readto record'
   /* place your task specific commands here */
   /* to modify the record                   */
   'output' record
end
error: exit RC*(RC¬=12)       /* return 0 if end of file */
```

Figure 54. Model of a REXX User-Written Stage Command

See the *z/VM: CMS Pipelines User's Guide and Reference* for an example of an Assembler user-written stage.

Chapter 23. Using the Batch Facility

The CMS batch facility provides a way of submitting jobs for batch processing in CMS. You can use the CMS batch facility when:

- You have a job (like an assembly or execution) that takes a lot of time, and you want to be able to use your terminal for other work while the time-consuming job is running.
- You do not have access to a terminal.

The CMS batch facility is really a virtual machine generated and controlled by the system operator. The operator logs on z/VM using the batch user ID and initiating the CMS batch facility by issuing one of the following:

- Issuing the BATCH parameter in the PARM field of the IPL command
- Issuing the NOSPROF parameter of the IPL command and issuing CMSBATCH when the VM READ status appears.

All jobs submitted for batch processing are spooled to the user ID of this virtual machine, which sequentially executes the jobs. To use the CMS batch facility at your location, you must ask the system operator what the user ID of the batch virtual machine is.

Submitting Jobs to the CMS Batch Facility

Under a real OS or DOS system, JCL specifications control jobs submitted in batch mode. Batch jobs submitted to the CMS batch facility are controlled by the control cards /JOB, /SET, and /*, and by CMS commands.

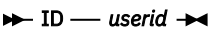
Any application or development program written in a language that z/VM supports can be executed on the batch facility virtual machine. However, there are restrictions on programs using certain CP and CMS commands, as described later in this section.

Input to the Batch Machine

Input records must be in card-image format and may be punched on real cards, placed in a CMS file with fixed-length, 80-character records, or punched to your virtual punch. These jobs are sent to the batch virtual machine in one of two ways:

- By reading the real punched card input into the system card reader, or
- By spooling your virtual punch to the virtual reader of the batch virtual machine.

When you submit a real card deck to the batch machine, the first card in the deck must be a CP ID card. The ID card takes the form:



`ID` must begin in card column one, and *userid* is the batch facility virtual machine user ID.`ID` and *userid* must be separated by one or more blanks.

For example, if the batch virtual machine for your installation has a user ID of BATCH1, punch the following card and place it in front of your deck:

```
ID BATCH1
```

When you are going to submit a job using your virtual punch, you must first be sure that your punch is spooled to the virtual reader of the batch virtual machine. For example:

```
spool punch to batch1
```

Batch Considerations for Shared File System (SFS) Files

It is not recommended that you use the CMS batch facility for files that contain SFS commands or SFS Callable Services Library (CSL) routines. However, if you choose to do so, you should use the following guidelines to ensure that your jobs are properly executed.

When you submit a job to the batch machine, it connects to the SFS file pool server by using the user ID of the batch machine. With the batch machine user ID, the application checks the authorization of the files and directories it must access to run your job. Therefore, if you want to submit jobs to refer to SFS files or directories, you must issue commands to authorize the batch machine to access them. The batch machine must also be enrolled in your file pool (or if an ENROLL PUBLIC command was issued, the batch machine can connect).

Note: Some batch facilities may be set up to run batch jobs under the user ID of the virtual machine submitting the job. Your system administrator can tell you if the batch facility you are using is set up that way. If it is, the batch machine runs your job under your user ID. For example, if you place an explicit lock on a file or directory, the job running on the batch machine can still use that file. However, if your batch job has a file open for update, any job you are running (from your user ID) will be barred from updating the file until the batch job is finished. Note that the batch facility provided with z/VM does not run under the user ID of the virtual machine submitting the job, but rather under its own user ID.

Attention: You should be aware that by authorizing the batch machine to access files and directories, you may be creating a security problem. Other batch machine users can connect to your file pool and use your files or directories.

Because batch jobs use the user ID of the batch machine, it is important that you explicitly state your user ID in ACCESS commands. If you use a period (.) to refer to your user ID in an SFS directory, your batch job may fail. For example, if you (YOURID) submit a batch job that accesses the PUBS:YOURID.SALES directory with a file mode of B, you must state the command as follows:

```
access pubs:yourid.sales b
```

If, instead, you specify:

```
access pubs:.sales b
```

The batch machine substitutes its own user ID, (for example, BATCH1) for the user ID. Because the directory will be identified incorrectly, the batch job will fail, or if `batch1.sales b` exists, you will get unexpected results.

Therefore, if you are writing an application or exec that may eventually be run on a batch machine, be sure to specify the user ID.

The same problem may occur if you write an application or exec that allows the file pool ID to default. The default will not be the default file pool ID of the submitting machine. Instead, it will be the default (if one is defined) for the batch machine. Therefore, if you want to allow the file pool ID to default, you should also submit a SET FILEPOOL command with your batch job.

Finally, if you submit a batch job with FILEWAIT on and the resource is being used, your batch job and all other batch jobs in queue will wait until the required resource is freed. This could conceivably tie up the batch facility for the maximum job limit of up to 131,068 seconds. Therefore, it is suggested that you do not set FILEWAIT on when submitting jobs to the CMS batch facility.

Submitting Virtual Card Input to the CMS Batch Facility

Virtual card input can be spooled to the batch machine in several ways. You may create a CMS file that contains the input control cards and use the CMS PUNCH command to punch the virtual cards:

```
punch batch jcl (noheader
```

When you punch a file this way, you must use the NOHEADER option of the PUNCH command, since the CMS batch facility cannot interpret the header card that is usually produced by the PUNCH command. As it does with cards in an incorrect format, the batch virtual machine would erase the header card.

You can use an EXEC procedure to submit input to the batch machine. From an EXEC, you can punch one line at a time into your virtual punch, using the EXECIO command. When you do this, you must remember to issue the CP CLOSE command to release the spool punch file when you are finished. For example:

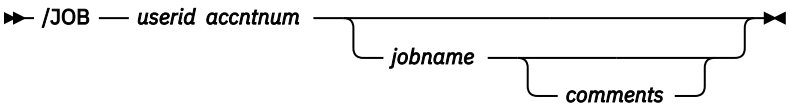
```
close punch
```

If you are using the exec to punch individual lines and entire CMS files to be read by the batch virtual machine as one continuous job stream, you must remember to spool your punch as follows:

```
/* EXEC to submit a batch job to CMS BATCH */
'SPOOL PUNCH CONT TO BATCH3'
'EXECIO 1 PUNCH (STRING /JOB MCGUIRE 999888'
'PUNCH BATCH JCL * (NOHEADER'
'SPOOL PUNCH NOCONT CLOSE'
```

/JOB and /* Cards

A /JOB card must precede each job to be executed under the batch facility. It identifies your user ID to the batch virtual machine and provides accounting information for the system. It takes the form:



/JOB must begin in card column one.

userid
is your user identification or the user ID under which you want the job submitted. This parameter controls:

- The user ID charged by the CP accounting routines for the system resources used during a job.
- The name and distribution code that appear on any spooled printer or punch output. Any spooled output will be spooled under the user ID specified.
- The user ID to whom status messages are sent while the batch machine is executing the job.

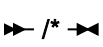
Note: The first and second items are correct only if the directory for the batch virtual machine involved contains the accounting option.

accntnum
is your account number. This account number appears in the accounting data generated at the end of your job. It overrides the account number in the CP directory entry for the user ID specified for this job.

jobname
is an optional parameter that specifies the name of the job being run. If you specify *jobname*, it appears as the CP spool file identification in the file type field. The file name field always contains CMSBATCH. See "Batch Facility Output" on page 363 for more information.

comments
may be any additional information you want to provide.

The /* card indicates the end of a job to the batch facility. It takes the form:



/* must begin in card column one. The batch facility treats all /* cards after the first as null cards. Therefore, if you want to ensure against the previous job not having a /* end-of-job indicator, you should precede your /JOB card with a /* card.

The /* card is also treated as an end-of-file indicator when a file is being read from the input stream. This is a special technique used in submitting source or data files through the card reader and is discussed under “Batch Exec for a Non-CMS User” on page 366.

Note:

1. Both "/JOB" and "/*" must begin in column 1.
2. The /* card can contain only the characters "/*". No other characters can appear on this input card.

/SET Card

The /SET card sets limits on a system's time, printing, and punching resources during the execution of a job. It takes the form:

/SET must begin in card column one.

seconds
is a decimal value that specifies the number of virtual CPU seconds the job can use.

lines
is a decimal value that specifies the maximum number of lines a job can print.

cards
is a decimal number that specifies the maximum number of cards a job can punch.

The default values for the batch facility are set at 131,068 seconds, printed lines, and punched cards per job. Any new limits defined using the /SET card must be less than these maximum settings. The system resources can be set at lesser values than the default values by an installation's system programmer; be sure you know the maximum installation values for batch resource limits before you use the /SET card.

A /SET card appears anywhere in the job following the /JOB card. The new limits defined by the /SET card apply only to the part of the job that follows the /SET card.

A job can contain up to three /SET cards (one for each operand); a /SET card cannot be entered more than once for the same operand.

Only use /SET cards for the operands whose values you want to change from the default; the default values are reset between jobs. A /SET card for an operand overrides its default but does not reset the other operands.

The /SET card for CMSBATCH is designed to set an approximate limit on the amount of resources a single batch job can use so that one job will not exclude other jobs from a resource. The actual amount of resource used is compared to the limit set at a logical checking point (for example after a buffer has been printed when using the PRINT command). Because the comparison is done at a logical checking point, the maximum specified may have been exceeded at any time since the last comparison check. The job will stop at the point when a comparison is done and the specified limit is reached or exceeded.

Note:

1. "/SET" must begin in column 1.
2. Programs can issue the STIMERM macro without interfering with batch time limits. If STIMER REAL is issued, the CMSBATCH time limit will never expire.

Other Input Records

The remainder of input records in the batch job consist of CP and CMS commands that are entered. (For a description of command restrictions, see “Restrictions on CP and CMS Commands in Batch Jobs” on page 362). EXEC, EXEC 2, or REXX statements cannot be imbedded in the input stream.

How the Batch Facility Works

The CMS batch facility, once initialized, runs continuously. When it begins executing a job, it sends a message to the user ID of the user submitting the job. If you are logged on when the batch machine begins executing a job that you sent it, you receive the message:

```
MSG FROM batchid:  JOB yourjob STARTED
```

When the batch machine finishes processing a job, it sends the message:

```
MSG FROM batchid:  JOB yourjob ENDED
```

yourjob is the job name you specified on the /JOB card. Before it reads the next job from its card reader, the batch virtual machine:

- Closes all spooling devices and releases spool files
- Resets any spooling devices identified by the CP TAG command
- Detaches any minidisk devices that were accessed
- Punches accounting information to the system
- Reloads CMS.

All of this is done by the CMS batch facility so that each job that is executed is unaffected by any previous jobs.

If a job that you sent to the batch virtual machine abnormally terminates (abends), the batch machine sends you the following message:

```
MSG FROM batchid:  JOB yourjob ABEND
```

The batch machine also spools a CP storage dump of the batch virtual machine to the printer. Then, the batch virtual machine stops processing your job and the job is flushed.

Whenever the batch virtual machine has read and executed all of the jobs in its reader, it waits for more input.

Preparing Jobs for Batch Execution

When you want to submit a job to the CMS batch facility for execution, you should provide the same CMS and CP commands you would use to prepare to execute the same job in your own virtual machine.

You must provide the batch virtual machine with read access to any input files that are required for the job. You do this by supplying the LINK and ACCESS commands necessary. The batch virtual machine has a minidisk at virtual address 195 with file mode A, so you can issue commands to access your minidisks or SFS directories as read-only extensions. For example, if you wanted the batch machine to execute a program module named LONDON on your 291 minidisk, your input file might contain the following:

```
/JOB FISH 012345
CP LINK MCGUIRE 291 291 RR SECRET
ACCESS 291 B/A
LONDON
```

Similarly, if you are using the batch virtual machine to execute a program using input and output files, you must supply the file definitions:

```
CP LINK ARDEN 391 391 RR FOREST
ACCESS 391 B/A
FILEDEF INFILE DISK VITAL STAT B
FILEDEF OUTFILE PUNCH
CP SPOOL PUNCH TO MCGUIRE
LONDON
```

If you expect printed or punched output from your job, you may need to include the spooling commands necessary to control the output. In the previous example, the punch of the batch machine is spooled to the virtual machine associated with user ID MCGUIRE.

Any output printer files produced by your job are spooled by the batch virtual machine to the printer. **These files are spooled under your user ID and with the distribution code associated with your user ID, provided the batch machine's directory has the accounting option set**. You can change the characteristics of these output files with the CP SPOOL command. For example:

```
cp spool e class t
```

If you want output to appear under a name other than your user ID, use the FOR operand of the SPOOL command. For example:

```
cp spool e for jonson
```

Output punch files are spooled, by default, to the real system card punch (under your user ID), unless you issue a SPOOL command in the batch job to control the virtual card punch of the batch virtual machine.

Note: If you are using the batch machine for files stored in SFS directories, you may not want to set FILEWAIT on. If your batch job is accessing several files in different directories, the job could wait too long and hold up other jobs.

Restrictions on CP and CMS Commands in Batch Jobs

The batch facility permits many CP and most CMS commands.

Note: If a batch job uses PIPE to issue a CP command, the batch machine will process the command. It will not restrict the execution.

The following CP commands are permitted and can be used to control the batch virtual machine:

```
CHANGE          MSG
CLOSE           QUERY
DEFINE          REWIND
DETACH          SMSG
DUMP            SPOOL
DISPLAY         STORE
LINK            TAG
LOADVFCB
```

These CP commands are subject to the following restrictions in batch jobs.

- CHANGE, CLOSE, and SPOOL cannot be used to affect devices RDR or 00C.
- DETACH can not be used to affect devices RDR, PUN, PRT, 00C, 00D, or 00E.
- Do not use the CHANGE, CLOSE, and SPOOL commands to affect the virtual reader.
- Do not use the DETACH command to detach any spooling devices or the system, IPL, or 19E disks.
- You must enter the LINK command on one line in the following format:

  ```
  CP LINK userid vaddr1 vaddr2 mode password
  ```

  You cannot use the LINK command keywords “To”, “As”, or “PASS=”.

  The *userid* cannot be “TO” or “T”. The *vaddr2* cannot be “A”; you can use “0A”. The *password* cannot be “PASS=”.

  If the minidisk does not have a password associated with it, the *password* must be “ALL”.

  A maximum of 26 links can be in effect at any one time.
- If a DIAGNOSE code X'08' is issued, the CHANGE and SPOOL commands will have an effect on the virtual card reader.
- Preface all CP commands in a batch job with **CP**.

- If you are running a CMS, EXEC 2, or REXX exec in the batch environment, the return code may be different than if the same exec is run in CMS. BATCH intercepts some commands and checks their validity for the CMSBATCH environment. The return code may be from BATCH and not from CP.

The following restrictions apply to CMS commands used in batch jobs:

- Because the batch virtual machine reads input from its reader, do not use the following commands or operands that affect the reader:

```
ASSGN SYSxxx READER (CMS/DOS only)
DISK LOAD
FILEDEF READER
READCARD
RECEIVE
```

- Jobs running under the CMS batch machine should not use the RDCARD macro.
- Do not use the following CMS SET command operands:

```
    AUTODUMP        LOADAREA
    AUTOREAD        OUTPUT
    BLIP            PROTECT
    IMESCAPE        REDTYPE
    IMPCP           RELPAGE
    INPUT           SERVER
    KEYPROTECT      STORECLR
```

  All of the other CMS SET command operands can be used in a job executing in the batch virtual machine. However, if the SET TRAPMSG command is invoked with the STOP option while in batch mode, the STOP option will be ignored.
- Concatenation of commands by using the new line character (usually an X'15') results in an error message.

Batch Facility Output

Any files that you request to have printed during the execution of your job are spooled to the real system printer under your user ID, unless you have spooled it otherwise. Once released for processing, these output files are under the control of the CP spooling facilities; if you are logged on, you can control the disposition of these files before they are printed with the CLOSE, PURGE, ORDER, and CHANGE commands. See “Purging and Reordering Batch Jobs” on page 366.

Output files produced by the batch virtual machine are identifiable by the file name CMSBATCH in the CP spool file name field. The spool file type field contains the file type JOB, unless you specified a jobname on the /JOB card. This applies to both printer and punch output files.

In addition to your regular printed output, the CMS batch facility spools a console file that contains a record of all the lines read in, and the responses, error messages, and return codes that resulted from command or program execution. The file name and file type of this spool file is BATCH CONSOLE.

Using Exec Files for Input to the Batch Facility

There are a variety of ways that exec procedures can help facilitate the submission of jobs to the CMS batch facility. The following examples are very simple. You probably would not go to the trouble of sending such a job to the batch virtual machine for processing. The examples do, however, illustrate the two basic ways that you can use exec procedures with the batch facility:

- Initiating an exec procedure from a batch virtual machine
- Using an exec procedure to create a job stream for the batch virtual machine.

Example 1: You can prepare an exec file that contains all of the CMS commands you want to execute, and then pass the name of the exec to the batch virtual machine.

Suppose you have the following file, COPY JCL:

```
/JOB CARBON 999999
CP LINK MCGUIRE 191 391 RR LINKPASS
```

```
ACCESS 391 B/A
EXEC COPYF
/*
```

and the file, COPYF EXEC:

```
/*    */
'COPYFILE FIRST FILE B SECOND = A'
'COPYFILE THIRD FILE B FOURTH = A'
```

If you enter the commands:

```
spool punch to cmsbatch
punch copy jcl * (noheader
```

the commands in COPYF EXEC are executed by the batch virtual machine.

Example 2: You could also use an exec to punch input to the batch virtual machine. Using the same commands as in the previous example. Suppose you have the following exec named BATCOPY that contains the same commands used in the previous example:

```
/* exec to submit a batch job */
'CP SPOOL PUNCH TO BATCH3 CONT'
punch = 'EXECIO 1 PUNCH (STRING'  /* initializing a variable */
punch '/JOB CARBON 999999'
punch 'CP LINK MCGUIRE 191 391 RR LINKPASS'
punch 'ACCESS 391 B/A'
punch 'COPYFILE FIRST FILE B SECOND = A'
punch 'COPYFILE THIRD FILE B FOURTH = A'
punch '/*'
'CP SPOOL PUNCH NOCONT CLOSE'
```

Then, when you enter the exec name:

```
batcopy
```

the input lines are punched to the batch virtual machine.

In either case, the execs you use may be simple or complicated. In the first instance, an exec might contain many steps, with control statements to conditionally control execution, error routines, and so on.

In the second instance, you might have an exec that is versatile so that it can be used with different arguments so as to satisfy more than one situation. For example, you can create a simple exec to send any job to the batch virtual machine to be assembled. This exec, called BATCHASM EXEC, might contain:

```
/* An exec for batch assemblies */
'CP SPOOL PUNCH TO BATCH3 CONT'
arg filename .
punch = 'EXECIO 1 PUNCH (STRING'  /* initializing a variable */
punch '/JOB MCGUIRE 888888'
punch 'CP LINK MCGUIRE 191 391 RR LINKPASS'
punch 'ACCESS 391 B/A'
punch 'ASSEMBLE' filename '(PRINT'
punch 'CP SPOOL PUNCH TO MCGUIRE'
punch 'PUNCH' filename 'TEXT A'
punch '/*'
'CP SPOOL PUNCH NOCONT CLOSE'
```

Now, whenever you want the CMS batch facility to assemble a source file for you, enter:

```
batchasm filename
```

The batch virtual machine will assemble the source file (*filename*), print the listing, and send you a copy of the resulting TEXT file.

Sample System Procedures for Batch Execution

To extend the previous example a little further, suppose you wanted to process source files in languages other than the assembler language. You want, also, for any user to be able to use this exec. You might have a separate exec file for each language and an exec to control the submission of the job. Figure 55 on

page 365 shows the controlling file and BATCH EXEC, and Figure 56 on page 365 shows the language file and ASSEMBLE EXEC.

```
/* This exec submits assemblies/compilations to CMS Batch
     - Punch batch job card
     - Call appropriate language exec to punch executable commands */

    arg accnt filename language
    if language = ''
      then do
        say 'Correct form is: BATCH accnt filename language'
        exit 100
      end
    punch = 'EXECIO 1 PUNCH (STRING'
    trace errors
    signal on error
    'CP SPOOL D CONT TO BATCHCMS'
    punch '/JOB' Userid() accnt filename
    punch 'CP LINK' Userid() '191 291 RR SECRET'
    punch 'ACCESS 291 B/A'
    'EXEC' language filename Userid()
    punch 'RELEASE 291'
    punch 'CP DETACH 291'
    punch '/*'
    'CP SPOOL D NOCONT CLOSE'
    exit
    Error: exit rc
```

Figure 55. BATCH EXEC

```
/* Correct form is : ASSEMBLE fname userid
     Punch commands to:
       - Invoke CMS assembler
       - Return text deck to caller        */

    arg fname userid
    signal on error
    trace errors
    punch = 'EXECIO 1 PUNCH (STRING'
    punch 'GLOBAL MACLIB UPLIB DMSGPI OSMACRO'
    punch 'CP MSG' userid 'Asmbling' fname
    punch 'ASSEMBLE' fname '(PRINT NOTERM)'
    punch 'CP MSG' userid 'Assembly done'
    punch 'CP SPOOL D TO' userid 'NOCONT'
    punch 'PUNCH' fname 'TEXT A1'
    punch 'CP CLOSE D'
    punch 'CP SPOOL D OFF'
    exit
    Error: exit rc
```

Figure 56. ASSEMBLE EXEC

Suppose FAY (user ID = FAY) invokes the BATCH EXEC by issuing the following command:

```
batch 1111 payroll assemble
```

The BATCHCMS virtual machine's reader should contain the following statements (in the same general form as a FIFO console stack):

```
/JOB FAY 1111 PAYROLL
CP LINK FAY 191 291 RR SECRET
ACCESS 291 B/A
GLOBAL MACLIB UPLIB DMSGPI OSMACRO
CP MSG FAY Asmbling PAYROLL
ASSEMBLE PAYROLL (PRINT NOTERM)
CP MSG FAY Assembly done
CP SPOOL D TO FAY NOCONT
PUNCH PAYROLL TEXT A1
CP CLOSE D
CP SPOOL D OFF
RELEASE 291
CP DETACH 291
/*
```

When the batch facility executes this job, the commands are executed as you see them: if you are logged on, you receive, in addition to the usual messages that the batch facility issues, those messages that are included in the exec.

Batch Exec for a Non-CMS User

Many installations run the CMS batch facility for non-CMS users to submit particular types of jobs. Usually, a series of exec files are stored on the system disk so that each user only needs include a card to use the exec, which executes the correct CMS commands to process data included with the job stream.

For example, if a non-CMS user wanted to compile FORTRAN source files, the following BATFORT EXEC file could be stored on the system disk:

```
/* EXEC for batch FORTRAN Compiles */
arg filename
'FILEDEF INMOVE TERM (RECFM F BLOCK 80 LRECL 80'
'FILEDEF OUTMOVE DISK' filename 'FORTRAN A1 (RECFM F LRECL 80 BLOCK 80'
'MOVEFILE'
'GLOBAL TXTLIB VSF2FORT'
'FORTVS2' filename '(PRINT)'
fortret = rc
if rc = 0 then
  'PUNCH' filename 'TEXT A1'
exit fortret
```

To use this exec, a non-CMS user might place the following real card deck in the system card reader:

```
ID CMSBATCH
/JOB  JOEUSER 1234 JOB10
BATFORT  JOEFORT
.
source file
.
/* (end-of-file indicator)
/* (end-of-job indicator)
```

When the batch virtual machine executes this job, it begins reading the exec procedure and executes one line at a time. When it encounters the MOVEFILE command, it begins reading the source file from its card reader (the batch facility interprets a terminal read as a request to read from the card reader). It continues reading until it reaches the end-of-file indicator (the /* card) and then resumes processing the exec on the next line following the MOVEFILE command.

Additional functions may be added to this exec procedure, or others may be written and stored on the system disk to provide, for example, a compile, load, and execute facility. These exec procedures would let an installation accommodate the non-CMS users and maintain common user procedures.

Purging and Reordering Batch Jobs

If you are logged on to the batch user ID, you can control the execution of batch virtual machine jobs when required by purging, reordering, and restarting them; by the same token, because all the closed printer files are queued for system output under the submitting user ID, you can change, purge, or reorder these files before processing on the system printer.

To purge a job executing under the batch monitor, use the following procedure:

1. Signal attention so VM READ appears in the status area of the batch machine console.
2. Issue the HX (halt execution) Immediate command.
3. Disconnect the virtual machine using the #CP DISCONN command. # is the default line end character.

The HX command causes the batch facility to abnormally terminate. This provides the user with an error message and a CP dump of the batch facility virtual machine. The batch monitor then loads itself again and starts the next job (if any).

The batch monitor normally runs with the console spooled NOTERM. Therefore, if you are logged on to the batch user ID, and you want to see the current job stream execution (if any) or the results of any of the following commands, issue a #CP SPOOL CONS TERM.

To purge an individual input spool file that is not yet executing, enter the following CP PURGE command:

```
#cp purge reader spoolid
```

spoolid is the spool file number of the job to be purged from the job queue of the batch virtual machine.

To reorder individual spool files in the job queue of the batch facility, enter the following CP ORDER command:

```
#cp order reader spoolid1 spoolid2...
```

spoolid1 and *spoolid2* are the assigned spool file identifications of the jobs to be reordered.

You can determine which jobs are in the queue by entering the following CP QUERY command:

```
#cp query reader all
```

This QUERY command lists the file names and file types of all the jobs in the batch virtual machine's job queue. You can then reorder them using the ORDER command.

Chapter 24. Creating an Interactive Program

This chapter describes two dialog management systems:

- The Interactive System Productivity Facility (ISPF)
- The Display Management System for CMS (DMS/CMS).

Your application can communicate with the terminal by writing one line at a time and reading one data item at a time. However, applications using several data items are greatly simplified by using ***dialogs*** between the user and the computer.

A common way to create dialogs is using full-screen displays or ***panels***. Although it is possible to create panels as data areas in programs and write them to the terminal one line at a time, this uses a lot of storage and is time-consuming. Also, the task of dialog management itself—that is, controlling the flow from one panel to the next—can be very complex. That is why it is better to use a standard data communications interface or ***dialog management system***.

ISPF provides services that complement standard z/VM services and that implement interactive processing. The Display Management System for CMS (DMS/CMS) provides a way to implement interactive processing in z/VM. DMS/CMS lets you design panels that can be displayed from applications written in REXX, EXEC 2, CMS EXEC, COBOL, PL/I, RPG II, or assembler. DMS/CMS is a simple and fast panel generation tool used in application development. It may be used by any customer with VM application development requirements as an alternative to ISPF.

Using ISPF for Dialogs

The Interactive System Productivity Facility (ISPF) provides services that lets you create interactive applications. As an application programmer, you can use ISPF to:

- Display messages or predefined full-screen images (panels)
- Originate and maintain tables of user information
- Generate output files to be processed by other applications
- Define and control symbolic variables
- Control the various kinds of operational modes during processing
- Interface to Edit and Browse facilities (using ISPF/PDF).

An application that runs under ISPF is called a dialog. You can code your dialog in various programming languages. You can even use more than one language in a dialog. There are also facilities that let you use REXX.

Each dialog is made up of various programs and data elements. There are five types of dialog elements, some of which are optional. These are the three most commonly used elements:

- **Functions** are command procedures or programs that perform processing requested by you, such as display of panels and messages, building of tables, generation of output files, and control of operational modes.
- **Panels** are predefined display images, such as menus, data entry panels, and information-only panels.
- **Messages** are comments that provide special information to you, such as confirmation that a user-requested action is in process or completed, or a report of an error in the user's input.

There are two elements that are not as commonly used:

- **Tables** are two-dimensional arrays used to maintain data. Tables can be temporary or retained across sessions and shared among several applications.
- **File Tailoring Skeletons** are generalized images of sequential data that can be customized during a dialog to produce an output file.

Panel definitions, message definitions, and skeletons are stored in libraries before execution of the dialog. You create them by editing directly into the panel, message, or skeleton libraries. No compiling or preprocessing step is required. Tables are generated and updated during dialog execution. Functions are programs or sequences of commands that you write to invoke and control the various ISPF elements and services.

The following sections show you how to use ISPF to develop a dialog. See Appendix J, "ISPF Example," on page 599 for a complete FORTRAN program using ISPF.

Developing an ISPF Dialog

To develop a dialog, you use an editor to enter the various components. You can use either XEDIT, or the edit option of the ISPF/Program Development Facility (ISPF/PDF).

You create panels by editing a file panel, defining the panel by keywords and options, and then saving the file as a member of an ISPF library.

A panel definition closely resembles the 3270 screen image that appears when the panel is displayed. Each character position in the panel definition is mapped to the same relative position on the display screen. You control where variable and literal data appear by entering the variable name or literal itself on the panel definition exactly where you want it to appear.

You create messages in the same way, but they are saved in a message library. Each member of a message library can contain several messages, each one referenced by a unique message ID. You specify the message text, the name of the corresponding HELP panel (to be displayed if the user requests help when the message is displayed), and an indication whether an audible alarm will be sounded. You can also specify a short message text to be displayed in the upper right-hand corner of the screen or some other position you specify.

You also create functions with the editor. Your FORTRAN program, for example, can invoke ISPF services by calling an ISPF service interface routine called ISPLNK. On the call statement, you describe the services required. For example, suppose you have a panel called USRNAME in your panel library.

To display USRNAME from a FORTRAN program, code:

```
INTEGER DSPSRV(2),PANEL(2)
DATA    DSPSRV/'DISP','LAY '/
DATA    PANEL/'USRN','AME '/
     .
     .
     .
LASTRC=ISPLNK(DSPSRV,PANEL)
```

The same panel can be displayed from a REXX EXEC by using Address ISPEXEC:

```
Address ISPEXEC 'DISPLAY(USRNAME)'
```

How to Begin Using ISPF

To use ISPF, certain requirements must be met. First of all, ISPF must be available to you, usually by means of a CMS system disk such as the S-disk or the Y-disk. If you are not sure where the ISPF licensed program resides, ask your supervisor. Each installation can install ISPF to suit their own needs, which can vary considerably. You will need the various libraries distributed with the ISPF licensed program.

The ISPF libraries distributed are:

ISPPLIB MACLIB
Panel Libraries

ISPMLIB MACLIB
Message Libraries

ISPTLIB MACLIB
Table Input Libraries

You also need the ISPSTART command to begin dialog processing. If these commands and libraries are not available to you, consult your supervisor or system administrator.

Before you invoke ISPF, your virtual device 191 must be accessed as the A-disk. During operation, ISPF assumes that this minidisk is always in read/write mode and that no other user has write access to it. (In some cases, ISPF permits multiple write access to minidisks other than 191, provided that such access is performed under the control of ISPF.)

The libraries distributed with ISPF are system libraries. To make these as well as your own libraries available to applications running under ISPF control, you need to issue some FILEDEF commands, which should remain in effect throughout your ISPF session. Suppose you have a panel library called USRPANEL, and a message library called USRMESGS. You need to concatenate these libraries with the corresponding distributed libraries, and you want your libraries accessed ahead of the distributed libraries. The next sequence of commands (which can be included in your PROFILE EXEC or in another exec) make these libraries available to ISPF functions:

```
FILEDEF ISPPLIB DISK USRPANEL MACLIB * (PERM CONCAT)
FILEDEF ISPPLIB DISK ISPPLIB  MACLIB * (PERM CONCAT)
FILEDEF ISPMLIB DISK USRMESGS MACLIB * (PERM CONCAT)
FILEDEF ISPMLIB DISK ISPMLIB  MACLIB * (PERM CONCAT)
```

Notice that the ddname in each pair of FILEDEFs is the same as the file name of the distributed ISPF library. Other ISPF libraries follow the same pattern:

ISPTLIB
is the ddname for all the table input libraries.

There are four optional libraries that are user-defined:

Skeleton library
ddname ISPSLIB

Table Output library
ddname ISPTABL

File Tailoring Output library
ddname ISPFILE

Profile library
ddname ISPPROF

The PERM option ensures that the FILEDEF remains in effect throughout an ISPF session. The CONCAT option concatenates two or more libraries under the same ddname. The order that libraries are searched is the same as the order that the FILEDEFs are issued. (You do not have to issue a GLOBAL MACLIB command before invoking ISPF.)

If the ISPF commands and libraries are not on a system disk, but are available by means of the LINK command, you might want to write an EXEC to link the ISPF system disk and issue the FILEDEFs you need. If the ISPF system disk is on a minidisk with a virtual address of 591, owned by a user called ISPMAINT, with a read password of ALL (that is, not requiring a password to link), the following statements in REXX do this:

```
/* ACCESS ISPF SYSTEM */
'CP LINK ISPMAINT 591 591 RR'
'ACCESS 591 Z/A'
'FILEDEF ISPPLIB DISK USRPANEL MACLIB * (PERM CONCAT)'
'FILEDEF ISPPLIB DISK ISPPLIB  MACLIB * (PERM CONCAT)'
'FILEDEF ISPMLIB DISK USRMESGS MACLIB * (PERM CONCAT)'
'FILEDEF ISPMLIB DISK ISPMLIB  MACLIB * (PERM CONCAT)'
```

Note: See Chapter 20, “Using Execs,” on page 333 for a discussion of execs.

You can create panel and message libraries by using XEDIT together with the MACLIB command. Create each panel with the editor first, then build the panel library with the MACLIB command.

Note: The panels and groups of messages must have a CMS file type of COPY. When using the editor to create a panel and to specify a file type of COPY, be sure to enter the editor subcommand SERIAL OFF to prevent the editor from inserting serial numbers in the panel file in columns 73 - 80. If these numbers are

present, they will cause ISPF errors. You can also use a different file type (for example, PANEL or MSG) and then rename the file before building the library.

The following steps outline a method of building a panel or message library:

1. XEDIT MENUPAN PANEL
2. (Create Panel)
3. FILE MENUPAN COPY
4. XEDIT NAMEPAN PANEL
5. (Create Panel)
6. FILE NAMEPAN COPY
7. MACLIB GEN USERPAN MENUPAN NAMEPAN.

In steps 1 and 4, the panel members are created by using a file type of PANEL to bypass serialization. In steps 2 and 5, edit subcommands create the panel members. In steps 3 and 6, a form of the FILE subcommand is used to write the files to disk with a file type of COPY. In step 7, the MACLIB command is used to create USERPAN MACLIB. This library contains the two members MENUPAN COPY and NAMEPAN COPY.

After you create the panels and messages you need, you can develop an application using REXX, or you can develop your specific language application.

Once your programs are compiled and exist either as text or load modules, you need to make them available to ISPF by issuing the appropriate FILEDEF command. For example, if you write a program called TESTPROG and compile it, you have a file called TESTPROG TEXT A1 on you A-disk. If you want to include TESTPROG in a TXTLIB called DEVLIB TXTLIB A1, issue the TXTLIB GEN or TXTLIB ADD command to insert the TEXT file into the library. This command makes the library available to ISPF:

```
FILEDEF ISPXLIB DISK DEVLIB TXTLIB * (PERM)
```

If you have included the module in a LOADLIB, use:

```
FILEDEF ISPLLIB DISK DEVLIB LOADLIB * (PERM)
```

When a text module is invoked (either as a TEXT file or as a member of a TXTLIB), any other text modules that it calls are loaded automatically by automatic call reference. The modules must also be TEXT files on a ISPF-accessible minidisk or members of the TXTLIB allocated to ddname ISPXLIB. If you have more than one TXTLIB, use the CONCAT option of the FILEDEF command to concatenate the libraries under the same ddname, ISPXLIB.

If your program is in a LOADLIB, use the ddname ISPLLIB. You can also specify a concatenated sequence for LOADLIBs. No automatic call referencing occurs with load modules. All load module references must be resolved before invocation by ISPF. Load modules can be used only for programs that are reenterable.

Note: Avoid using nonrelocatable files whenever possible. User nonrelocatable files can create a very complex operational environment, because CMS subset mode is turned on to prevent nonrelocatable files from overlaying relocatable programs already in storage. When using split screen mode, CMS subset mode is not turned off until all relocatable programs associated with logical screens have completed execution.

When you have created the dialog functions you need, you can invoke the ISPF environment by means of the ISPSTART command, using the appropriate PANEL, CMD, or PGM parameter.

- The PANEL parameter causes the panel specified to be displayed, and passes any options to it that are specified on the ISPSTART command line.
- The CMD parameter specifies the name of an exec to be invoked as the first dialog function.
- PGM specifies the program name to be invoked as the first dialog function.

ISPF Dialog Organization

You can organize dialogs in a number of ways to suit the needs of the application. A typical dialog, for example, starts with a display of the highest menu in a hierarchy. This is the ***primary option menu***. User options selected from this menu can invoke a dialog function, or display a lower level menu. The lower level menu can also cause functions to receive control, or pass control on to still other lower level menus. This hierarchical organization (tree structure) might look like Figure 57 on page 373.

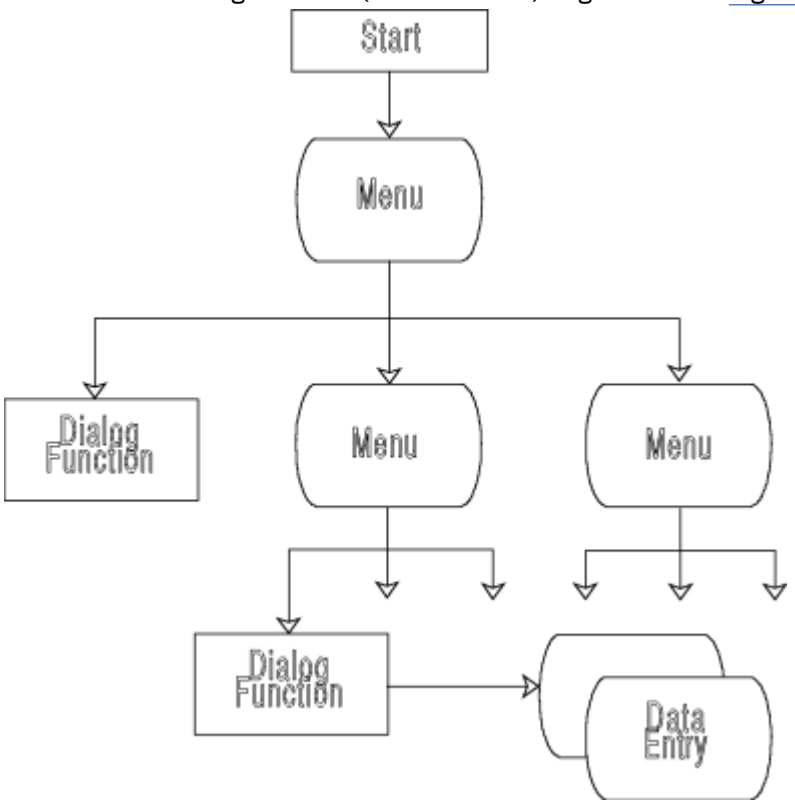


Figure 57. A Typical Dialog Starting with a Menu

Eventually, a dialog function receives control. When it does, it can use any of the dialog services provided by ISPF, including panel display for data entry. When the function completes execution, control is passed back up the tree to the panel from which the function was selected. Control eventually returns to the primary option menu. The process can now begin again with a different dialog path.

Controlling Dialog Flow with the SELECT Service

Your first major task in developing a dialog application is to design the dialog itself. That is, you have to define the structure and flow of panels, services and functions that make it up. Controlling the flow in a dialog is made possible by the SELECT service. ISPF uses the SELECT service during its initialization to invoke a function or selection panel that begins a dialog. During dialog processing, SELECT can be used to display menus and invoke program or command procedure functions.

The same parameters used on the ISPSTART command line (PANEL, CMD, and PGM) can be passed to the SELECT service to specify the next action to be taken. If the CMD parameter is used, the exec it invokes can in turn invoke other execs, without requiring use of the SELECT service. When the PGM parameter is specified, the function it invokes can call other programs, which are considered part of the same function. If you call a function from within a program, use the SELECT service. Figure 58 on page 374 illustrates how the SELECT service invokes and processes a dialog.

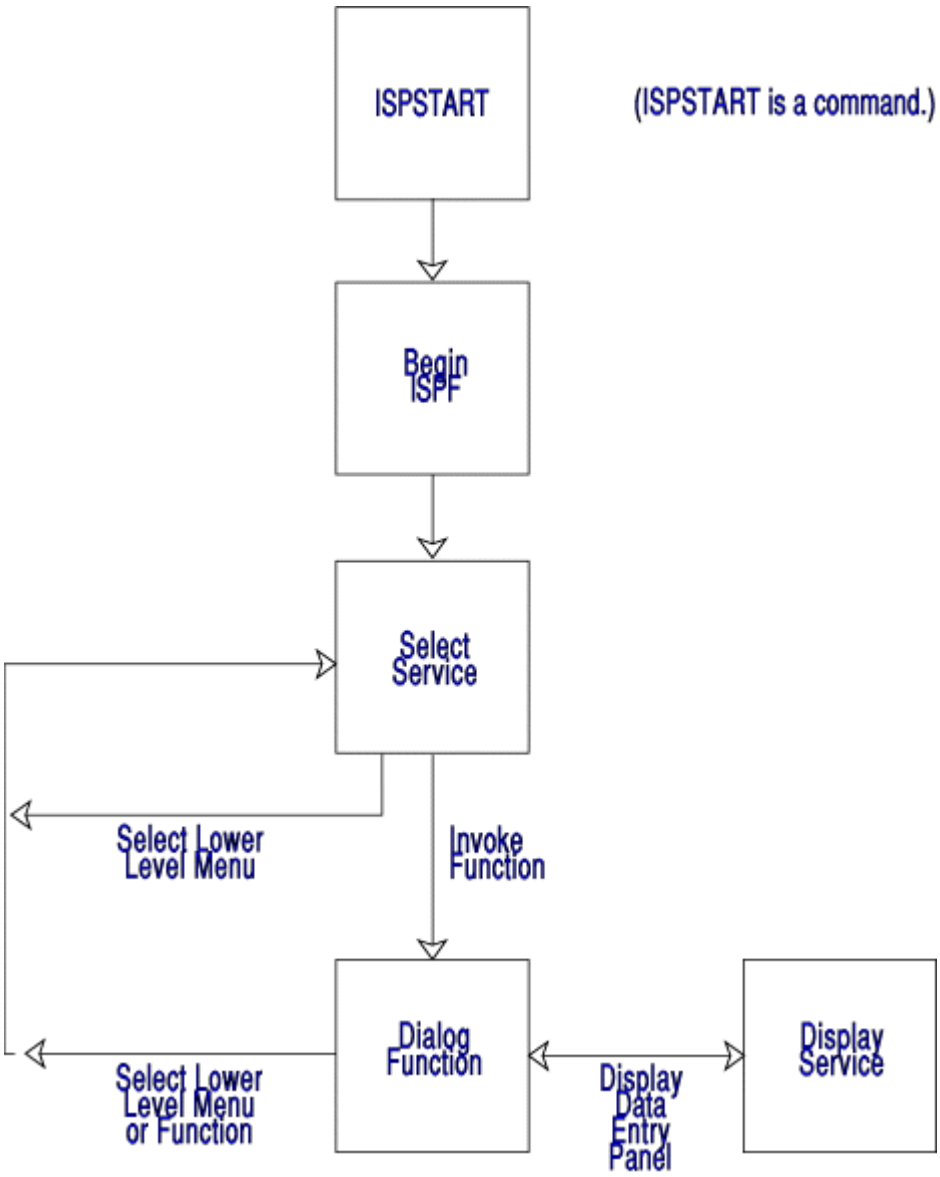


Figure 58. SELECT Service Used to Invoke and Process a Dialog

ISPF Panel Definition

You define a panel in ISPF using up to seven sections, of which only two (the BODY and END sections) are required for all panels. The PROC section is required for all selection panels. The seven sections are:

1. **Attribute section** defines the special characters used in the body of the panel definition to represent attribute (start-of-field) bytes, such as high intensity, low intensity, and input field.
2. **Body section** defines the format of the panel as seen by the user, and defines the name of each variable field on the panel.
3. **Initialization section** specifies the processing that will occur before the panel is displayed. You usually use this section to define how variables are to be initialized.
4. **Reinitialization section** specifies the processing that will occur before redisplay of a panel.
5. **Processing section** specifies the processing that will occur after the panel is displayed. You usually use this section to define how variables are verified and translated.
6. **Model section** (required for table display only; not allowed for other types of panels) specifies the format for displaying each row of the table.

7. **End section** consists of only the)END statement. ISPF ignores any data that appears on lines following the)END statement.

The panel display service recognizes these default field attribute characters:

+
text (protected) field, low intensity

%
text (protected) field, high intensity

–
input (unprotected) field, high intensity.

Each panel definition section begins with a statement that indicates the section being defined. There are seven statements, one for the start of each of the sections. The statements are:

)ATTR
attribute section

)BODY
body section

)INIT
initialization section

)REINIT
reinitialization section

)PROC
processing session

)MODEL
model section (table displays only)

)END
end of panel definition

You can define all data entry panels of a dialog using only the)BODY and)END statements and the default field attributes. The following screen definition does not contain the other statements.

```
)BODY
%---------------------------  EMPLOYEE RECORDS  -----------------------------
%COMMAND ===>_ZCMD
%
%EMPLOYEE SERIAL: &EMPSER
+
+   TYPE OF CHANGE%===>_TYPECHG  +  (NEW, UPDATE, OR DELETE)
+
+   EMPLOYEE NAME:
+     LAST   %===>_LNAME          +
+     FIRST  %===>_FNAME          +
+     INITIAL%===>_I+
+
+   HOME ADDRESS:
+     LINE 1 %===>_ADDR1                                  +
+     LINE 2 %===>_ADDR2                                  +
+     LINE 3 %===>_ADDR3                                  +
+     LINE 4 %===>_ADDR4                                  +
+
+   HOME PHONE:
+     AREA CODE   %===>_PHA+
+     LOCAL NUMBER%===>_PHNUM    +
+
)END
```

Figure 59. Sample ISPF Panel Definition

When this panel is displayed to the user, it looks like this:

```
--------------------------  EMPLOYEE RECORDS  --------------------------
COMMAND ===>

EMPLOYEE SERIAL:

   TYPE OF CHANGE ===>             (NEW, UPDATE, OR DELETE)

   EMPLOYEE NAME:
     LAST    ===>
     FIRST   ===>
     INITIAL ===>

   HOME ADDRESS:
     LINE 1  ===>
     LINE 2  ===>
     LINE 3  ===>
     LINE 4  ===>

   HOME PHONE:
     AREA CODE    ===>
     LOCAL NUMBER ===>
```

Figure 60. Sample ISPF Panel, When Displayed

For detailed information on how to define panels, see the *ISPF/PDF Services*.

ISPF Message Definition

You create message definitions using an editor, such as the System Product Editor. They are saved in a member of the message library. As with panel definitions, no compilation is required. Each message in the message library consists of two lines. The first line contains the message ID (required), short message text (optional), name of corresponding HELP panel (optional), and audible alarm indicator (optional). The second line contains the full message text.

The following message definitions contain all the fields:

```
MSG001   'OPERATION COMPLETED'     .HELP=MSGOK01   .ALARM=YES
'THE OPERATION SPECIFIED HAS BEEN COMPLETED.'
MSG002   'INVALID OPERATION'       .HELP=MSGNG01   .ALARM=YES
'ENTER A NUMBER FROM 1 TO 5 IN THE SPACE PROVIDED.'
```

If you want the first message to be issued during a dialog, you refer to the message by the identifier MSG001. The panel MSGOK01 can be invoked by the user with the HELP service. When the message is displayed, the audible alarm sounds. Finally, a short form of the message is provided for display in the upper right hand corner of a panel, in case you do not want the full message displayed right away.

ISPF Variable Definition

Variable services let you define and use ***dialog variables***. Dialog variables are the main communication vehicle between dialog functions (program modules or execs) and ISPF services. Program modules, execs, panels, messages, tables and skeletons can all reference the same data through the use of dialog variables.

The value of a dialog variable is a character string from zero to 32KB long. Some services restrict the length of dialog variable data; you can control the valid length of any dialog variable during panel and function definition.

You reference dialog variables by name. The name is composed of 1 to 8 characters, depending on the language you are using. Alphanumeric characters (A-Z, 0-9, #, $, or @) can be used in the name, but the first character must be nonnumeric. In the sample panel definition shown earlier the names TYPECHG, LNAME, FNAME, I, ADDR1, ADDR2, ADDR3, and ADDR4 are all names of dialog variables.

If you write a function in a language like FORTRAN, identify the internal variables to be used as dialog variables to ISPF with the ISPF variable service VDEFINE. The program can also access and update dialog variables using VCOPY and VREPLACE. These services do not apply to execs.

ISPF Panel Services

You can use two ISPF panel services to manipulate panels.

DISPLAY
displays data entry panels.

SELECT
displays a hierarchy of selection panels (menus).

The DISPLAY service controls the display of individual panels, such as data entry, informational, or HELP panels. The SELECT service is used in a dialog to create a hierarchy of functions and menus that determine the sequence in which those functions and menus are processed.

The DISPLAY service reads a panel definition from the panel library, initializes variable panel fields from corresponding dialog variables, and displays the panel on the screen. A message can also be displayed with the panel.

The user can enter information in fields specified on the panel definition as input fields. After the user presses Enter, the contents of the input fields are stored in dialog variables specified on the panel definition. Then, any processing specified on the panel definition using the)PROC statement is performed. The DISPLAY service returns to the calling function. Optionally, the cursor can be positioned at the start of any field in the panel definition.

For example, in a FORTRAN program, to display a panel called USRNAME, plus a message in the message library called PERX110, and to position the cursor at the field called LNAME, use the following calling sequence:

```
INTEGER DSPSRV(2),PANEL(2),PRX110(2),CRX110(2)
DATA    DSPSRV/'DISP','LAY '/
DATA    PANEL/'USRN','AME '/
DATA    PRX110/'PERX','110 '/
DATA    CRX110/'LNAM','E   '/
  .
  .
  .
LASTRC=ISPLNK(DSPSRV,PANEL,PRX110,CRX110)
```

From a REXX exec, the command is:

```
Address ISPEXEC 'DISPLAY PANEL(USRNAME) MSG(PRX110) CURSOR(CRX110)'
```

You can also use the DISPLAY service to display messages, independently of panels. Do this by omitting the PANEL parameter; this causes the)REINIT section to be processed, and the current panel is overlaid with the message specified in the MSG parameter.

If you do not specify the panel-name or message-id, the)REINIT section is processed, and the current panel is redisplayed without a message.

You use the SELECT service to display and control a hierarchy of selection panels. Menus (selection panels) make up a special class of panels. A menu must have an input field to be used for the entry of selection options by the user of the application. This field, the standard name of which is ZCMD, is usually the first input field on line two of the panel. Corresponding to the ZCMD variable there must be a processing section in the panel definition in which ZCMD is translated and stored in the variable ZSEL. ISPF uses ZSEL as input to the SELECT services. This parameter can be used to select a still lower panel definition. In this way, a path from the primary option menu can be defined down to the lowest level.

ISPF Variable Pools

To maintain multiple levels of control, dialog variables are organized into groups called ***variable pools***, according to the dialog function and application with which they're associated.

A variable pool is basically a list of variable names that lets ISPF access the associated variables. When an ISPF service encounters a dialog variable name (for example, in a panel or message table), it searches these pools to access the value of the dialog variable. There are three types of variable pool:

Function pool
contains variables only accessible by a given function.

Shared pool
allows functions and selection panels to share access to dialog variables.

Shared pools are created by the SELECT service when it processes the ISPSTART or ISPF command and when the NEWAPPL or NEWPOOL keywords are specified with the SELECT service. When SELECT returns, the shared pool is deleted and the previous shared pool (if any) is reinstated.

Application profile pool
contains variables retained for the user from one ISPF session to another. Profile variables are automatically available when an application begins and are automatically saved when it ends.

ISPF Variable Services

Many services are available in ISPF to control dialog variables:

VGET
retrieves variables from a shared pool.

VPUT
updates variables in a shared pool or profile pool.

VDEFINE
defines function variables.

VDELETE
removes definition of function variables.

VRESET
resets function variables.

VCOPY
copies data from a dialog variable to the program.

VREPLACE
copies data from the program to a dialog variable.

VGET and VPUT can be invoked from any function. The other variable services are for use from program modules only.

Like the panel and message services, you can invoke variable services from a FORTRAN program, for example, using ISPLNK. You can use the following FORTRAN statements to invoke the VDEFINE service. The statements define the function variable LNAME and initialize it:

```
IMPLICIT INTEGER (A-Z)
DIMENSION LNAME(4)
LASTRC = ISPLNK('VDEFINE', '(LNAME)',LNAME, 'CHAR',16)
```

(LNAME) is the name of the function variable. LNAME is the field to contain the value of the variable function. LNAME is initialized to spaces. CHAR is the literal "CHAR", which indicates the format of the variable. The length of the variable field, 16 bytes.

Other ISPF Services

Other services are available in ISPF for dialog management. You can invoke each service from a program as shown for ISPLINK (COBOL) or ISPLNK (FORTRAN).

Table Services

ISPF table services let you maintain and access sets of dialog variables. A table is a 2-dimensional array of information in which each column corresponds to a dialog variable. Each row contains a set of values for those variables.

A table can be either temporary or permanent. Temporary tables exist only in virtual storage and can't be written to disk storage. Permanent tables are created in virtual storage, but can be saved on disks.

File Tailoring Services

Another type of ISPF service is the file tailoring service. These services read skeleton files from a library and write tailored output that can be used to drive other functions. The file tailoring output can be directed to a library, a sequential file, or both that has been specified by the ISPF function. It can also be directed to a temporary sequential file provided by ISPF.

Each skeleton file is read record-by-record. Each record is scanned to find any dialog variable by name. When a variable name is found, its current value is substituted from a variable pool.

The file tailoring services are:

FTOPEN
prepares the file tailoring process. It specifies whether the temporary file will be used for output.

FTINCL
specifies the skeleton to be used, and starts the tailoring process.

FTCLOSE
ends the file tailoring process.

FTERASE
erases (deletes) an output file created by file tailoring.

Miscellaneous Services

In addition to display, variable, table, and file tailoring services, ISPF provides EDIT, BROWSE, LOG, and CONTROL services.

The EDIT and BROWSE services are available only if PDF is installed. These services let you invoke the PDF edit or browse programs from a dialog function, specifying a CMS file.

The LOG service lets a dialog function write a message to the ISPF log file, which can be used as an audit or tracking mechanism.

The CONTROL service lets a dialog function condition ISPF to expect certain kinds of display output, or to control the disposition of errors encountered by ISPF services. The CONTROL service lets you:

- LOCK the terminal keyboard during a display
- Split a display screen if required (or inhibit screen splitting)
- REFRESH the entire screen on the next display
- Permit panels to be processed without displaying them.

Error-handling CONTROL parameters lets you terminate the dialog function upon receipt of a return code of 12 or higher (CANCEL parameter), or to RETURN control to the dialog function on all errors.

Using DMS/CMS for Dialogs

The Display Management System for CMS (DMS/CMS) provides a way to implement interactive processing in z/VM. DMS/CMS lets you design full screen images (called panels) that can be displayed from applications written in COBOL, PL/I, RPG II, assembler, REXX, EXEC 2, or CMS EXEC.

DMS/CMS has three functional parts:

1. **Panel Formatter**

Panel designers use the Panel Formatter to design the content and format of panels (DMS/CMS calls designed screens panels). The word 'screens' is used to mean the DMS/CMS interactive screens. Panel designers use a display terminal to design panels. There are only a few basic DMS/CMS rules a panel designer must follow. Generally, anyone who can use a display terminal can design panels.

2. **Panel Manager**

   Programmers use the Panel Manager to associate their COBOL, PL/I, RPG II, and assembler application programs or their REXX, EXEC 2, or CMS EXEC procedures with defined panels. Programmers must know how to write programs in the language they choose to use with the Panel Manager.

3. **Write Full Screen**

   Assembler programmers may use the Write Full Screen to take advantage of DMS/CMS's full-screen I/O capability to a 3270-type graphics device. While this part can be very useful to a assembler programmer, it can be ignored by most DMS/CMS users.

   Note: DMS assumes that it is the only full screen panel manager running at the time of use. If DMS panels are being used with another full screen editor or writer application, it may be necessary to release the DMS panels and reset the DMS environment and buffers. DMS is not recommended to be used with any other full screen editor or writer at one time.

DMS/CMS Users

There are three tasks to consider when talking about how to use DMS/CMS.

1. **Application End Use**

   The end user enters information on the DMS/CMS panels. This person needs to know how to use a display terminal.

2. **Panel Designing**

   The panel designer used DMS/CMS to lay out the panels to be used by the application end user. The panel designer must know how to use a display terminal and follow the DMS/CMS rules. The end user and the panel designer may be the same person.

3. **Programming**

   The programmer writes the application program or EXEC procedure that works with the designed panels. The programmer needs to know how to use the programming language or EXEC procedure being employed. This person may also be the panel designer and a user.

Programmers use the Panel Manager to associate their programs or EXEC procedures with the designed panels. They may use COBOL, PL/I, RPG II, or assembler to write programs; REXX, EXEC 2, or CMS EXEC to write EXEC procedures.

The following table lists the features of DMS/CMS for each of the tasks described above.

Table 66. Features of DMS/CMS

Task	DMS/CMS Features
Application End Use	• Interactive use of display terminal • Extended Highlighting • Color • Wide screen usage • Selector light pen usage

Table 66. Features of DMS/CMS (continued)	
Task	**DMS/CMS Features**
Panel Designing	• Interactive test of panels without programming • Ability to change delimiters that identify fields • Display of field type and number • Can issue PANEL command with panel name as operand • Can define multiple panels with one call • Can override the automatic cursor skip at the end of field
Programming	• REXX, EXEC 2, CMS EXEC, Assembler, COBOL, PL/I, and RPG II • Can position cursor anywhere on displayed panel • User's cursor position is returned to program • Can dynamically change protection for data fields • Can control how data entered on a displayed panel is passed to the program

System Support Functions

Applications using DMS/CMS services may execute on the 24-bit or 31-bit architecture machines.

DMS/CMS calls operation system services using standardized interfaces so that DMS/CMS is affected less during operating system release migrations.

Panel Formatter Functions

The panel designer uses the Panel Formatter to design panels that the end user uses. Among the functions DMS/CMS offers to help in panel design are:

1. Interactive formatting on a number of different display terminals, including large-screen, wide-screen, and color terminals.
2. Ability to define types of fields (data, text, or light-pen selectable) and to define the characteristics of the field. The possible characteristics include protected or not protected, alphanumeric or numeric, color, intensity, highlighting, or autoskip at field end. Programs may further modify the existing panel format.
3. Editing commands to help in panel design. Among the commands offered are those to copy or move lines, to duplicate lines, to delete lines, to center lines, to shift lines left or right, to scroll the display forward or backward, and to display the panel as the end user will see it.

Panel Manager Functions

The programmer uses the Panel Manager to associate a program with the designed panels. Among the functions DMS/CMS offers to help the programmer are:

- The ability to program in different languages.
- The ability to use EXEC procedures (REXX, EXEC 2, or CMS EXEC).
- Cursor control. The programmer can position the cursor anywhere on the displayed screen. The position of the user's cursor can be returned to the program.
- The ability to change field attributes dynamically.
- The ability to control how user-entered data is passed to the program (left or right justification and fill, and upper or mixed case).

Panel Size Considerations

DMS/CMS works with terminals having several different size screens. Terminal screens with widths of 80 or 132 characters and heights of 24, 27, 32, or 43 rows can be used with DMS/CMS.

A panel designer working on a terminal screen of one size can design a panel for a terminal screen of another size.

If a designer is working on a terminal screen smaller than the screen that the panel is to be used on, only part of the panel will be visible at any one time. DMS/CMS has features to assist a designer in this situation.

If a designer presses a program function (PF) key, DMS/CMS shows the other side of the panel. Thus, when designing a 132-character wide panel on an 80-character wide terminal screen, DMS/CMS displays columns 1-70 of the panel being designed on the Design Grid screen (the grid occupies the other 10 spaces). The designer presses the PF key and DMS/CMS displays columns 63-132. When the PF key is pressed again, DMS/CMS redisplays columns 1-70.

Similarly, when designing a panel for a 27-row terminal on a 24-row terminal, not all of the 27 rows will be visible at any one time. DMS/CMS provides commands to scroll the display forward or backward.

Figure 61 on page 382 depicts the example given above. It shows how a designer uses a terminal with a size of 80 columns by 24 rows to design a panel for a terminal size of 132 columns by 27 rows. Many other size combinations are possible, and they would be handled in a similar fashion.

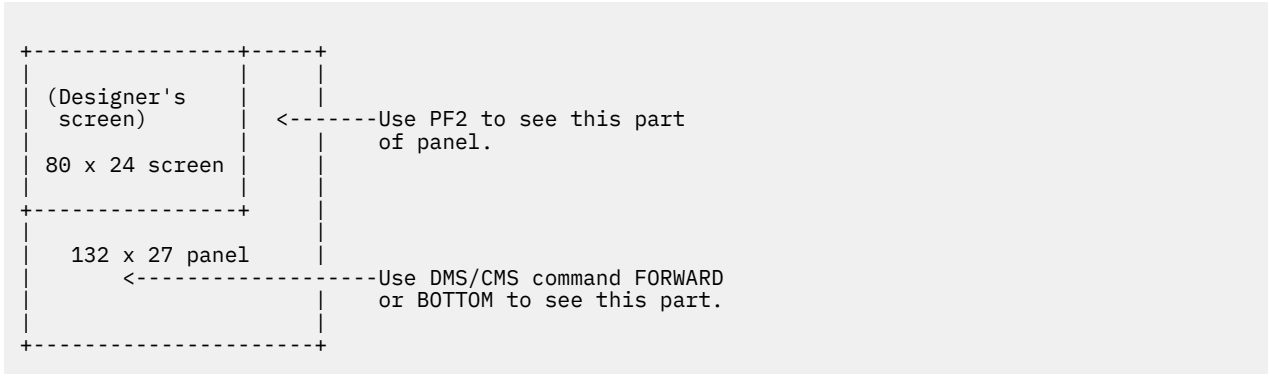


Figure 61. Designing a Panel for a Larger Size Terminal Screen

For more information, see the *Virtual Machine Display Management System for CMS, Version 2 Guide and Reference* and the *Virtual Machine Display Management System for CMS, Version 2 General Information*.

Chapter 25. Developing Commands Using the Parsing Facility

This chapter describes how to:

- Use the parsing facility, which consists of creating a definition language for command syntax (DLCS) file and issuing commands to process the DLCS file.
- Use the parsing facility from a DBCS language.
- Use the parsing facility from a REXX program and an Assembler program.
- Create your own CMS commands.

The CMS parsing facility parses and translates command arguments. Your programs can use the parsing facility to see if the user specifies the proper arguments on invocation and to see what the arguments are.

To use the parsing facility, you must define command syntax in a special language, the definition language for command syntax (DLCS). You keep the DLCS definitions in CMS files. A DLCS file can contain more than one DLCS definition. The parsing facility parses a specified command by checking whether all operands, options, keywords, and so on, are specified according to the DLCS definition for that command.

Defining command syntax in a DLCS file and using the parsing facility has the following advantages:

- Syntax checking is unnecessary in your program.
- All keyword abbreviations are expanded for you.
- Command syntax is defined separately from your program and can be translated into different national languages.
- When a national language is in use, keywords in that language are translated into the language recognized by your program.
- You do not have to write scanning code.
- The address and length of each token is provided.
- Validation codes are provided to identify the type of each token.

Note: If an application supplies system parser and synonym files (*xxx*SPA*cc* TEXT and *xxx*SSY*cc* TEXT, generated from a system DLCS file), application users can override and customize the application command syntax by creating user DLCS files. However, if an application supplies user parser and synonym files (*xxx*UPA*cc* TEXT and *xxx*USY*cc* TEXT, generated from a user DLCS file), application users can change the application command syntax only by modifying the DLCS source file and generating new TEXT files. System parser and synonym files must reside in a saved segment.

Using the Parsing Facility

To use the parsing facility, follow these steps:

1. Create a DLCS file.
2. Enter the GENCMD command with the CHECK option to check for any DLCS coding errors.
3. Enter the GENCMD command to put your syntax file into a machine readable form the parsing facility can use.
4. Enter the SET LANGUAGE command to enable the user's DLCS definitions.
5. Enter the PARSECMD command from a REXX program or EXEC 2 exec or the PARSECMD macro from an assembler program to invoke the parsing facility and to obtain the parsed and translated parameter lists.

Step 1. Creating a DLCS File

A DLCS file consists of:

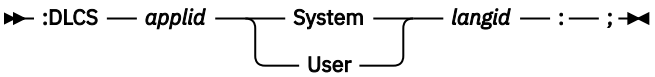
- A DLCS statement
- Command syntax definitions.

The DLCS file can have any file name. It can also have any file type not reserved for some other function. For a list of reserved CMS file types, see the *z/VM: CMS User's Guide*. The convention is to use a file type of DLCS, which is the default for input to the GENCMD command.

The DLCS Statement

The DLCS statement identifies the application where the commands in the DLCS file are parsed, to specify whether the commands are system or user commands, and to specify the national language for the file.

The format of the DLCS statement is:



applid
is an application identifier. It must be three alphanumeric characters, and the first character must be alphabetic (for example, DMS, HCP, OFS, AGW, DKK, and so on).

System|User
specifies whether the file contains system or user syntax definition statements. (Only the first letter is significant.)

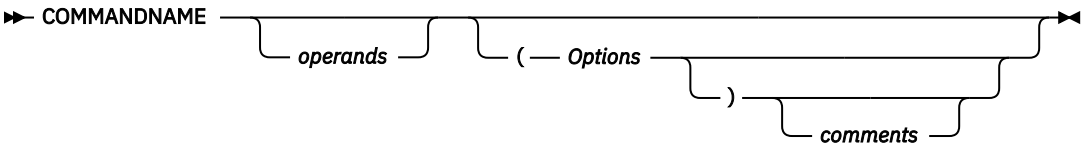
langid
is the identifier for the language you are working in. It must be one to five alphanumeric characters (for example, UCENG or AMENG). The language identifiers for supported languages are defined in the VMFNLS LANGLIST file.

Note:

1. The DLCS statement must be the first noncomment statement in the DLCS file, and it must be the only DLCS statement in the file.
2. The CMS command search order uses translations and translation synonyms defined in DLCS files with an application identifier of DMS.
3. The DLCS statement determines the file name and file type of the output files.

Command Syntax Definitions

Each command must be defined. Here is a standard CMS command string format:



DLCS has the following statements:

:CMD
for a command name

:OPR
for an operand

:OPT
for an option

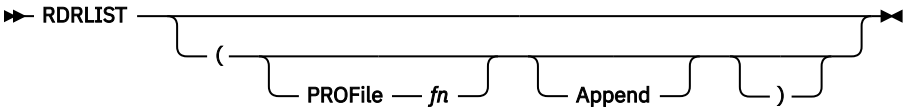
A few other statements you can use in DLCS include:

:SYN
to define synonyms

:KW.n
for command name modifiers

:*
to specify comments

For example, the RDRLIST command has the following format:



Here is how the syntax for RDRLIST is coded in DLCS:

```
:CMD D9K.RDRLIST RDRLIST RDRLIST 4 :;
  :SYN RLIST 2 :;
  :OPT KWL( <PROFILE 4> ) FCN(FN) :;
  :OPT KWL( <APPEND 1> ) :;
```

Step 2. Checking for DLCS Coding Errors

You can use the CHECK option of GENCMD to make sure your DLCS syntax descriptions are correct. In addition, you can issue GENCMD with the CHECK option while you XEDIT the DLCS file to help remove errors.

See the *z/VM: CMS Commands and Utilities Reference* for a description of GENCMD.

Step 3. Converting Your DLCS File

When all DLCS syntax errors are corrected, use the GENCMD command to convert the DLCS file into an machine readable form the parsing facility can use.

Step 4. Setting Command Name Synonyms and Translations

The SET TRANSLATE command sets user translation synonyms, user translations, system translation synonyms, and system translations on or off. The QUERY TRANSLATE command displays the contents of the system synonym tables, system translate tables, user synonym tables, and user translate tables. These commands work similarly to the CMS SYNONYM and QUERY SYNONYM commands.

(See the *z/VM: CMS Commands and Utilities Reference* for descriptions of SET TRANSLATE, QUERY TRANSLATE, SYNONYM, and QUERY SYNONYM.)

Step 5. Invoking the Parsing Facility

You can invoke the parsing facility in two ways:

1. Use the CMS PARSECMD command in an EXEC 2 or REXX exec. The name of the exec must be the CMS name of the command, as defined on the :CMD statement in the DLCS file.

   The PARSECMD command uses the EXECCOMM interface and creates exec variables that describe the translated command string. See Figure 64 on page 398 for an example using the PARSECMD command.

   See the *z/VM: CMS Commands and Utilities Reference* for a description of the PARSECMD command.
2. From assembler programs, use the PARSECMD macro.

   The PARSECMD macro call should be in the beginning of the program. Upon return from the parsing facility, the syntax of the command is verified and detailed information on the translated command string is available. See “Assembler Program Calling the Parsing Facility” on page 400 for an example using the PARSECMD macro.

See the *z/VM: CMS Macros and Functions Reference* for a description of the PARSECMD macro.

Command keywords are uppercased according to the national language uppercase table for the active application. If one is not found, the CMS national language table is used.

Coding Your Command Definitions

Define each command as follows:

- Specify the name of the command and its national language equivalent using a :CMD statement.
- Define any synonyms using :SYN statements immediately following the :CMD statement.
- Define a two word command using the first word as the command name and using the KW.1 statement to define the second word. If the command is a three word command, use the KW.2 statement to define the third word. (The second and third words are command name modifiers.) You can also have a four word command, a five word command, and so forth.
- Define the syntax for the command with zero or more :OPR statements followed by zero or more :OPT statements.
- Use the comment statement to add explanations to your DLCS file.

Rules to Remember

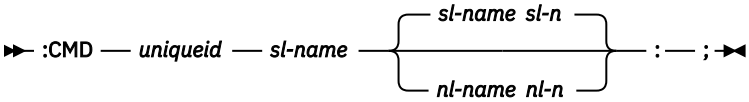
Some rules to remember while coding in DLCS are:

- Use special characters : , < > and ' in your data tokens (keyword names, function names, or function values) only if they are enclosed in single quotation marks. The quotation marks are not counted as part of the token.
- Use :; to specify the end of a statement.
- Do not use lowercase characters to specify your keyword names, function names, or function values (variables). Specify these exactly as they would appear after the command line is uppercased by the system at execution time with the language in effect.
- Only the first 72 characters of any line of the DLCS file are used. Any characters beyond 72 are ignored. You can use as many blanks as you want between tokens, and you can continue DLCS statements on the following line.
- Because only one system and one user DLCS file for an application can be active at any time, all command syntax definitions for an application are usually in one DLCS file. When both a user DLCS file and a system DLCS file have the same application ID name, the definitions in the user file override the definitions in the system file. If no user file is found for an application, the definitions in the system file are used.
- Your DLCS file must be merged with your user file for the application you currently have. You can have only one user table; therefore, if you have another command or receive a command from someone, you have to merge it with the other commands in the user table. (For example, if you want the CMS search order to find your command, define the command in a DLCS file with DMS application ID.)
- You can define the translation of some keywords to be the same as the keyword the command recognizes. For more information on translation, see the *z/VM: CMS User's Guide*.

Defining the Command Name Using the :CMD Statement

The :CMD statement defines the name of a command as the system sees it and as the national language sees it.

The format of the :CMD statement is:



uniqueid
identifies the syntax definition name for the command within the DLCS file. This is required, and it must be unique for each syntax definition. When you invoke the parsing facility, the *uniqueid* is matched to the *uniqueid* you specify on the PARSECMD command. These IDs must match exactly, that is even the uppercase and lowercase letters must match. For example, Anne=Anne but ANNE¬=Anne.

uniqueid is any combination of up to 16 characters. For quick access to the syntax definitions, the first one or two characters are used as an index. If the first two characters of *uniqueid* are valid hexadecimal digits, their value is used as the index. Otherwise, the EBCDIC value of the first character is used. For example, D9xxx and Rxxx have the same index value of 217. CMS can find syntax definitions faster if the unique IDs have different index values.

sl-name
is the command name as CMS sees it. The exec from which the PARSECMD command is called must be named *sl-name*.

nl-name
is the command name as a national language user sees it. The default is *sl-name*.

nl-n
is the minimum number of characters that must be entered for *nl-name* to be accepted. The default is *sl-n*, which is the length of *sl-name*.

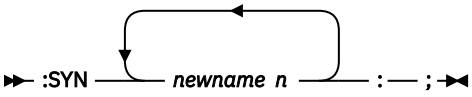
Note:

1. A new command syntax definition begins each time a :CMD statement is encountered.
2. All *uniqueid*s used for IBM commands have a period as the fourth character. Do not use a period as the fourth character in the *uniqueid* for your own commands.
3. A *uniqueid* of all blanks is reserved to let you define more than one translation for a command. When this *uniqueid* is found, no syntax information is stored. You can only code the :CMD and :SYN statements in this case.
4. The minimum length for abbreviations of command name translations cannot be more than eight or HELP does not recognize them.
5. The *nl-name* is only used by the CMS search order if the application identifier of this DLCS file is DMS.
6. The SET TRANSLATE command enables or disables *nl-name*.
7. If SET ABBREV OFF is in effect, you must use the full *nl-name*.

Defining Synonyms Using the :SYN Statement

The :SYN statement defines translation synonyms for the command nl-name defined on the :CMD statement.

The format of the :SYN statement is:



newname
the synonym you are assigning to the national language (nl) command name.

n
the minimum number of characters you must enter for the synonym to be accepted by CMS.

Note:

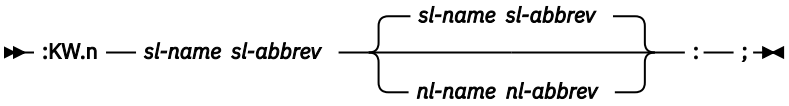
1. The :SYN statement is valid only for the first word of a command name (not the command name modifiers).
2. All of the :SYN statements for a command must immediately follow the :CMD statement.

3. Only :SYN statements defined in a DLCS file with an application identifier of DMS are used by the CMS command search order.
4. The SET TRANSLATE command enables and disables translation synonyms defined by the :SYN statement.
5. Using multiple names on a single :SYN statement has the same effect as specifying a single name on many :SYN statements. Order is not important.
6. If SET ABBREV OFF is in effect, you must use the full *newname*.

Defining Modifiers Using the :KW.n Statement

The :KW.n statement defines command name modifier keywords. These keywords modify the syntax used for parsing the remaining parameters. For example, a command to manipulate a simple database can require different operands—a file name for an open request, an option for a close request, and other operands for update requests. The :KW.n statement lets you define a different syntax for each.

The format of the :KW.n statement is:



n
is the number of the level. It defines the *n*th modifier after the command name.

sl-name
is the name of the command modifier keyword as the command sees it.

sl-abbrev
is the minimum number of characters that must be entered for *sl-name* to be accepted by CMS.

nl-name
is the name of the command modifier keyword as the national language user enters it. Defaults to *sl-name*.

nl-abbrev
is the minimum number of characters that must be entered for *nl-name* to be accepted by CMS. Defaults to *sl-abbrev*.

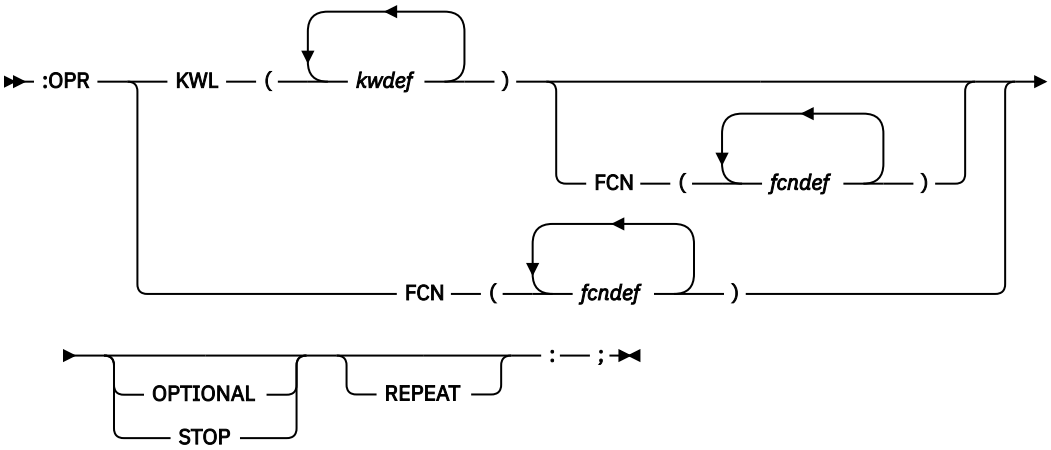
Use the following form of the :KW.n statement to indicate that a string of characters not defined by any :KW.n statement is accepted as an arbitrary modifier.

Note: This form may not be used as the first :KW.n statement on a level, and only applies to :KW.n statements on the same level. No further syntax information may follow this statement, that is, no :OPR, :OPT, or :KW.n statements with a larger value for n. When the parsing facility finds an arbitrary modifier it will process that remainder of the argument string as one text string as follows:

Defining Operands Using the :OPR Statement

The :OPR statement defines the syntax of each operand of the command.

The format of the :OPR statement is:



KWL
defines the operand as a keyword, *kwdef*. See "kwdef Expression" on page 390 for a description of the *kwdef* expression.

FCN
defines the operand as a function, *fcndef*. The value of *fcndef* is provided by the user; it is not a keyword. See "fcndef Expression" on page 390 for a description of the *fcndef* expression.

KWL FCN
defines the operand as a keyword-value pair using the *kwdef* and *fcndef* expressions. See "kwdef Expression" on page 390 and "fcndef Expression" on page 390 for a description of the *kwdef* and *fcndef* expressions.

OPTIONAL
indicates the operand can optionally be specified.

STOP
indicates that if the operand is not specified, then parsing of the operands stops at that point and no more operands can be specified.

REPEAT
indicates the operand can be specified one or more times.

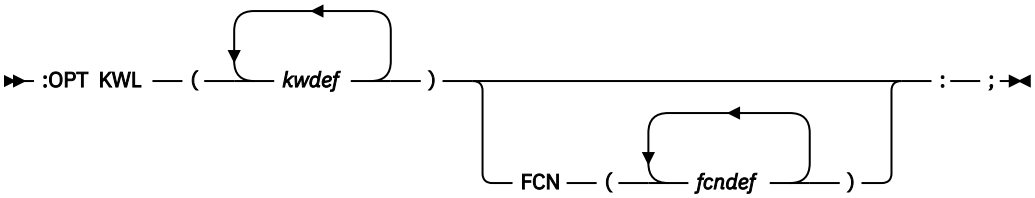
Note:

1. Specify :OPR statements in the order the operands are specified on the command.
2. Specify the :OPR statement after the :CMD statement and present :SYN statements or after appropriate :KW.n statements.
3. If both OPTIONAL (or STOP) and REPEAT are specified, the operand can be specified zero or more times.
4. If no options are specified, the operand is a required operand that can be specified only once.

Defining Options Using the :OPT Statement

The :OPT statement defines the syntax of the options for the command.

The format of the :OPT statement is:



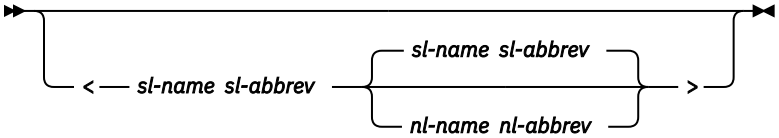
KWL
defines the option as a keyword, *kwdef*. See "kwdef Expression" on page 390 for a description of the *kwdef* expression.

KWL FCN
defines the option as a keyword-value pair using the *kwdef* and *fcndef* expressions. See "kwdef Expression" on page 390 and "fcndef Expression" on page 390 for a definition of the *kwdef* and *fcndef* expressions.

Note: :OPT statements must follow the last :OPR statement for the command. The order of the :OPT statements is not important.

kwdef Expression

The format of *kwdef* is:



sl-name
is the keyword known by your command.

sl-abbrev
is the minimum number of characters that must be entered for *sl-name* to be accepted.

nl-name
is the keyword known by a national language user. Defaults to *sl-name*.

nl-abbrev
is the minimum number of characters that must be entered for *nl-name* to be accepted. Defaults to *sl-abbrev*.

fcndef Expression

The *fcndef* expression can be a system function or a user function.

System Functions

Keyword
Valid input

ALPHANUM
any alphanumeric string

APPLID
any three character alphanumeric string with a the first character alphabetic

CHAR
any single nonblank character

CUU
any hex number between 001 and FFF (assumes leading zeros)

DIGITS
any unsigned number made up of digits 0-9

FN
(file name) any string with the following characters: A-Z,a-z,0-9,$,#,@,+,-,:, and _

FT
(file type) any string with the following characters: A-Z,a-z,0-9,$,#,@,+,-,:, and _

FM
(file mode) first character: A-Z, a-z; optional second character: 0-6

EFN
same as FN with '*' or '%' also a valid character

EFT
same as FT with '*' or '%' also a valid character

EXECNAME
any string that does not contain the following characters: =,*,(,),' ', and X'FF'

EXECTYPE
any string that does not contain the following characters: =,*,(,),' ', and X'FF'

DIRID
(SFS directory ID) any string that follows the rules for naming SFS directories as defined in the "General Concepts" chapter of the *z/VM: CMS Commands and Utilities Reference*. This function allows the user ID portion of the SFS directory ID to be specified as a user ID.

DIRIDN
(SFS directory ID) any string that follows the rules for naming SFS directories as defined in the "General Concepts" chapter of the *z/VM: CMS Commands and Utilities Reference*. This function allows the user ID portion of the SFS directory ID to be specified as either a user ID or a nickname.

FPOOLID
(file pool ID) any string with alphabetic (A-Z, a-z) and numeric (0-9) characters without imbedded blanks. The first character must be alphabetic. required.

NAMEDEF
any string with alphabetic (A-Z, a-z) and numeric (0-9) characters. The first character must be alphabetic.

HEX
any hexadecimal number

INTEGER
any decimal whole number (can have + or - signs)

INVALID
no valid values

INVFMDIR
invalid file mode or directory ID.

INVFMFPD
invalid file mode, file pool ID, or directory ID.

NINTEGER
any decimal negative whole number

PINTEGER
any decimal positive whole number (can have + sign)

PN
any BFS path name

MODE
any alphabetic character

STRING
any nonnull character string that does not include only blanks and does not include any X'5D' characters

TEXT
any character string

VDEV
any hex number between 0001 and FFFF

CSLPATH
the path number for a loaded CSL routine. It is a string consisting of two substrings separated by a period. Each substring is either an unsigned integer or an asterisk.

Note:

1. When you use :FCN to define the function of an operand or option, *fcndef* can be a group of valid values. Only items in the subset are valid. For example, if you specify STRING(MONDAY, TUESDAY, WEDNESDAY), MONDAY, TUESDAY, and WEDNESDAY are the only valid values.
2. If a list of functions is specified for *fcndef*, the parsing facility validates an operand or option value with the functions in the order they are specified. The first function the value is valid for determines the validation code of the value in PVCENTRY. See the *z/VM: CMS Macros and Functions Reference* for more information on the PVCENTRY macro.
3. Input to the parsing facility is uppercased according to current language before it is provided to system or user functions for validation.
4. If a value is not valid according to any of the functions in the list, the first one determines which message, if any, is issued. If an error message based on the first function is not appropriate, place the INVALID function first in the list. For example:

   ```
   :OPR FCN(INVALID, INTEGER(2,4,6), MODE) :;
   ```

   The INVALID function never accepts a value as valid, but a general error message is issued when a value is not valid according to the rest of the functions in the list.
5. Because some functions validate tokens that are also valid for other functions, you should be careful to list the most restrictive functions first. For example, an operand defined as:

   ```
   :OPR FCN(STRING, DIGITS, FN):;
   ```

   will always be validated as a string, while the syntax:

   ```
   :OPR FCN(FN) REPEAT:;
   :OPR FCN(DIGITS):;
   ```

   can never be satisfied because the required digits operand will be validated as part of the list of file names.
6. The TEXT function cannot be specified in a list with any other function.

User Functions

In addition to the system functions listed previously, you can also make your own functions for the parser to use to check if a token is valid. For instance, you could make a function VOWEL that considers only alphabetic characters A,E,I,O and U valid.

After you make your program for your function, assemble it, load it with the RLDSAVE option, and use the GENMOD command. Then install the MODULE file of this assembled program as a nucleus extension. Next, include the name of your function in the DLCS for your command exactly as you would any other function. The function is invoked by the parsing facility with a CMSCALL macro. The entry point name of the module must be the same as the function name (fcndef) in your DLCS file. Your function is passed the following parameters:

- An 8-byte area containing the function name.
- Token-addr: a fullword containing the address of the token to be validated. The token is already uppercased according to the current language.
- Token-length: a fullword containing the number of characters in the token.
- Validation code: a byte containing the number interpreted by the parser as the validation code of the user function. If the token is valid, this field should be set by the user function. Upon return from the parsing facility, you can check this validation code to see if your token is valid.

On entry to the program, R1 contains the address of the control block containing the parameters described previously. Use the assembler macro PARSERUF to generate a mapping of this control block.

Your program must pass back a return code in R15 that determines the outcome of the function. A return code of zero specifies the token was valid; a nonzero return code specifies the token was not valid. You

can use any nonzero return code except -3; this return code would be interpreted to mean the function did not exist.

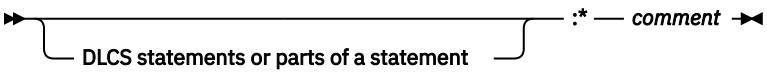
Note:

1. User functions do not override system functions with the same name (system functions come first in the search order).
2. When you use GENCMD to process your DLCS file, specify the ALL or USER options for user functions to be accepted.
3. When coding user functions in your DLCS file, you can enclose specific values in parentheses as you can with any system functions and only those values are accepted.

Writing Comments Using the :* Statement

The characters :* specify that a line or the remaining characters of a line are to be ignored. Use this to put comments and explanations in your DLCS file.

The format of a comment is:



comment
is any comment

For examples of creating a DLCS file, see "Creating the TEST DLCS File" on page 395 and "Creating the TEST DLCS File with Language Translations" on page 396.

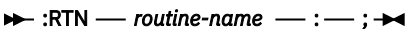
Defining Routines and Keywords Using the :RTN and :KWD Statements

Note: The :RTN and :KWD statements are reserved for IBM use. You may not use them in writing your own commands in DLCS. They are only shown here so that if you need to make your own translation of CMS commands you can do so without introducing errors into the syntax or its definition.

RTN Statements

The RTN statement defines the routine responsible for parsing the command.

The format of the RTN statement is:



routine-name
is a CMS defined name.

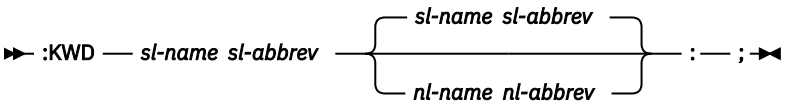
Note:

1. When the :RTN and :KWD statements are used, they replace (and are mutually exclusive with) the :OPR and :OPT statements. There is one :RTN statement followed by any number of :KWD statements.
2. When you are translating a CMS command that uses routine parsing, you should only change the *nl-name* and *nl-abbrev* fields on the :KWD statement. You must not add or delete :KWD statements or change the routine and system language names.

KWD Statements

The KWD statement defines the keywords that the command contains for translation purposes.

The format of the KWD statement is:



sl-name
is the CMS defined keyword name.

sl-abbrev
is the minimum number of characters that must be entered for *sl-name* to be accepted by CMS.

nl-name
is the keyword as a national language user enters it. Defaults to *sl-name*.

nl-abbrev
is the minimum number of characters that must be entered for *nl-name* to be accepted by CMS. Defaults to *sl-abbrev*.

Note:

1. When the :RTN and :KWD statements are used, they replace (and are mutually exclusive with) the :OPR and :OPT statements. There is one :RTN statement followed by any number of :KWD statements.
2. When you are translating a CMS command that uses routine parsing, you should only change the nl-name and nl-abbr fields on the :KWD statement. You must not add or delete :KWD statements or change the routine and system language names.

What the Parser Does Not Flag

1. The parser **does not** flag the following situations:
   - Dependent options and operands. The MAP operand of the MACLIB command gives an example. See the *z/VM: CMS Commands and Utilities Reference*.
   - Mutually exclusive options or operands. This is where you have a pair of operands or options. You must specify one or the other; you cannot specify neither or both. The ACK and NOACK operands of the NOTE command give an example. See the *z/VM: CMS Commands and Utilities Reference*. Most commands that have mutually exclusive options or operands ignore the condition and use the last operand or option you specify.
2. Some IBM supplied commands also use the RTN and KWD statements for special purposes. Do not use these statements for your own commands.

DBCS and the Parsing Facility

This section lists rules to remember when the current language is a double-byte character set (DBCS) language.

In DLCS and GENCMD

- You can use DBCS characters only in keywords, modifiers, and command names.
- You can mix single-byte and DBCS characters in a name in the DLCS, but GENCMD only recognizes single byte characters as DLCS delimiters.
- Shift-out and shift-in characters are always recognized as DBCS delimiters in a DLCS definition regardless of the current language.
- A double-byte character is treated as a single logical character. When you specify the minimum length for abbreviations of synonyms or translations, count double-byte characters and EBCDIC characters as single logical characters and ignore shift-out and shift-in characters. For example, if you have the keyword ‘abcd k1k2k3 efg’, setting the minimum abbreviation of four allows ‘abcd’ as the shortest abbreviation. Setting the minimum abbreviation of five, would allow ‘abcd k1 ’ as the shortest

abbreviation. Setting the minimum abbreviation of six allows ‘abcd k1k2 ’ as the shortest abbreviation, and so on.

- If you use DBCS characters when adding translations and translation synonyms to a DLCS file, you can enter GENCMD and SET LANGUAGE on these translations. However, you can only use these commands if the language you are using is set up as a double-byte language.

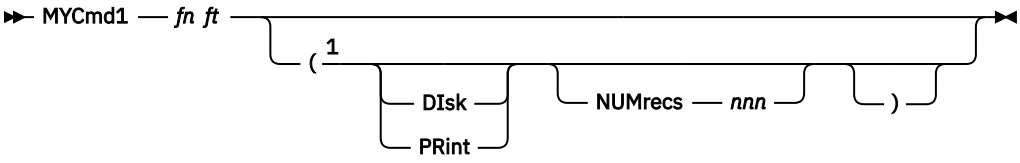
From CMS

- DBCS or mixed DBCS command names and keywords are accepted. DBCS strings cannot be specified for operand and option values such as file name, file type, file mode, cuu, and so on.
- Each token in the tokenized PLIST is resolved to be a complete DBCS string. In other words, one of these tokens can contain no more than three double-byte characters.
- When you invoke CMS commands, you can use DBCS EBCDIC to specify CMS delimiters such as blanks or parentheses.

Examples: Using the Parsing Facility

Suppose you have two commands, MYCMD1 and YOURCMD.

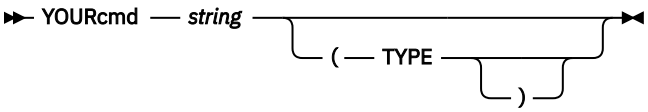
MYCMD1 has the following syntax:



Notes:

[1] When options are entered between the left '(' and right ')' delimiters, they can be in any order.

YOURCMD has the following syntax:



Instead of coding syntax checking into your program, you plan to invoke the parsing facility for these commands. Therefore, create a DLCS file, called TEST DLCS, to contain both syntax definitions.

Creating the TEST DLCS File

The TEST DLCS file contains the syntax definitions for MYCMD1 and YOURCMD commands. TEST DLCS might look like Figure 62 on page 395.

```
 1     :DLCS DMS USER AMENG :;
 2     :* The first command
 3       :CMD MMYCMD1 MYCMD1 MYCMD1 3 :;
 4         :SYN MY1 3 :;
 5         :OPR FCN(FN) :;
 6         :OPR FCN(FT) :;
 7         :OPT KWL(<DISK 2> <PRINT 2>) :;
 8         :OPT KWL(<NUMRECS 3>) FCN(PINTEGER) :;
 9     :* The second command
10       :CMD YYOURCMD YOURCMD YOURCMD 4 :;
11         :OPR FCN(STRING) :;
12         :OPT KWL(<TYPE 4>) :;
```

Figure 62. TEST DLCS File

where:

Line Number	Explanation
1	Defines this file for the DMS application, the commands as user commands, and the ID of the language as AMENG.
2	A comment indicating the start of the first command syntax definition.
3	Defines MMYCMD1 as the unique ID for this syntax definition, and MYCMD1 as the command name with a minimum abbreviation of MYC.
4	Defines a synonym, MY1, for the command name with no abbreviation.
5	Specifies the first required operand is a file name.
6	Specifies the second required operand is a file type.
7	Specifies two options: DISK as an option with a minimum abbreviation of DI, and PRINT as an option with a minimum abbreviation of PR.
8	Specifies another option as a keyword-value pair: NUMRECS as an option with a minimum abbreviation of NUM.
9	A comment indicating the start of the second command syntax definition.
10	Defines the unique ID and command name for this command definition.
11	Defines the only operand of this command as string.
12	Defines TYPE as the option with no abbreviation.

Creating the TEST DLCS File with Language Translations

You could also create TESTUCEN DLCS to contain national language translations for these two commands. If you wanted to include uppercase English translations, your file might look like this:

```
 1     :DLCS DMS USER UCENG :;
 2     :* The first command
 3       :CMD MMYCMD1 MYCMD1 UCENGMD1 8 :;
 4         :SYN MY1 3 :;
 5         :OPR FCN(FN) :;
 6         :OPR FCN(FT) :;
 7         :OPT KWL(<DISK 2 DISQUE 4> <PRINT 2 IMPRIMER 4>) :;
 8         :OPT KWL(<NUMRECS 3 NOMENREG 6>) FCN(PINTEGER) :;
 9     :* The second command
10       :CMD YYOURCMD YOURCMD VOTRECOM 5 :;
11         :OPR FCN(STRING) :;
12         :OPT KWL(<TYPE 4 AFFICHER 3>) :;
```

Figure 63. TESTUCEN DLCS File

where:

Line Number	Explanation
1	Defines this file for the DMS application, the commands as user commands, and the ID of the language to be UCENG.
2	A comment indicating the start of the first command syntax definition.
3	Defines MMYCMD1 as the unique ID for this syntax definition, MYCMD1 as the command name, and UCENGMD1 as the language name with no abbreviation.
4	Defines a synonym, MY1, for the command name with no abbreviation.
5	Specifies the first required operand is a file name.
6	Specifies the second required operand is a file type.

Line Number	Explanation
7	Specifies two options: DISK as an option with a minimum abbreviation of DI, and DISQUE as the national language name with a minimum abbreviation of DISQ. PRINT as an option with a minimum abbreviation of PR, and IMPRIMER as the national language name with a minimum abbreviation of IMPR.
8	Specifies another option as a keyword-value pair: NUMRECS as an option with a minimum abbreviation of NUM, and NOMENREG as the national language name with a minimum abbreviation of NOMENR.
9	A comment indicating the start of the second command syntax definition.
10	Defines the unique ID, command name, and national language name for this command definition.
11	Defines the only operand of this command as string.
12	Defines TYPE as an option with no abbreviation, and AFFICHER as the national language name with a minimum abbreviation of AFF.

Processing the TEST DLCS File

After you created the TEST DLCS file, you must use the GENCMD command with the CHECK option to make sure your syntax definition are correct. For example,

```
GENCMD TEST DLCS (CHECK
```

Once all syntax definition are correct, enter the following GENCMD command to convert your file into a format that can be read internally:

```
GENCMD TEST DLCS
```

Enter the following set language command to activate the language:

```
SET LANGUAGE (ADD DMS USER
```

Now, to invoke the parsing facility to process MYCMD1 see either “Processing MYCMD1 from a REXX Program” on page 397 or “Processing MYCMD1 from an Assembler Program” on page 399.

Processing MYCMD1 from a REXX Program

The following two sample REXX programs process MYCMD1. The first program, Figure 64 on page 398, illustrates a call to the parsing facility. The second program, Figure 65 on page 399, does not call the parsing facility; the exec performs all the syntax checking.

```
/*  MYCMD1 EXEC                                                 */
/*  processes the MYCMD1 command with this format:              */
/*       MYCmd1 fn ft ( DIsk|PRint NUMrecs nnn )                */
/* The options may be omitted; the file name and type           */
/* are required.                                                */
address command

/* First, call the parser to check syntax of the command        */
/* string.                                                      */
'PARSECMD MMYCMD1'

If rc ¬= 0 then signal error
       /* Go to ERROR if bad string.                           */
/* The command string is valid, so we can search through        */
/* the tokens to find out what options were specified.          */
/* It does not matter what language is active, because          */
/* the parser has translated the command name and any           */
/* options that were given.                                     */
/*                                                              */
/* We know that:                                                */
/*              token.1= the command name MYCMD1;               */
/*              token.2= the passed file name;                  */
/*              token.3= the passed file type;                  */
/* if it exists,token.4=OPTSTART;                               */
/* and if they exist, remaining tokens -.5, .6, .7 -            */
/* could be TYPE, DISK, NUMRECS, or nnn.                        */

how_to_output = 'DISK'          /* Set default output to        */
                                /* disk.                        */
number = '*'                    /* Set number to the            */
                                /* whole file.                  */
do i = 4 to token.0             /* Loop thru tokens, set        */
                                /* flags.                       */
  select
    when token.i = 'DISK' then how_to_output = 'DISK'
    when token.i = 'PRINT' then how_to_output = 'PRINT'
    otherwise                   /* Must be NUMRECS              */
      i = i + 1                 /* parameter.                   */
      number = token.i
  end
end

/* At this point, all of our flags and values have been         */
/* set, and we are ready to process the file.

. . .                                  */
```

Figure 64. MYCMD1: A REXX Exec Calling the Parsing Facility

```
/* This EXEC processes the MYCMD1 command with a format     */
/* as follows: MYCmd1 fn ft ( DIsk|PRint NUMrecs nnn )      */
/* The options may be omitted; the file name and type       */
/* address command                                          */
/* First lets see what was passed to us                     */
arg fn ft '(' options
if fn='' | ft = '' then signal NAME_MISS /*parms missing?   */
'ESTATE' fn ft '*'            /* check validity of fn/ft    */
if rc = 20 then signal NAME_ERROR /* incorrect fn or ft     */

how_to_output = 'DISK'    /* Set default output to    */
                          /* disk.                    */
number = '*'              /* Set number to the        */
                          /* whole file.              */
do while options ¬= ''   /* loop thru all options    */
  parse var options opt options
  select;
    when (opt='DI'|opt='DIS'|opt='DISK')
      then how_to_output = 'DISK'
    when (opt='PR'|opt='PRI'|opt='PRIN'|opt='PRINT')
      then how_to_output = 'PRINT'
    when (opt='NUM'|opt='NUMR'|,
          opt='NUMRE'|opt='NUMREC'|opt='NUMRECS')
      then do              /* verify validity of number     */
             parse var options num options
             if num = '' then signal NO_NUM_ERROR
             if ¬datatype(num,w) then signal INV_NUM_ERROR
             if num <= 0 then signal INV_NUM_ERROR
             number = num
           end
    otherwise                    /* unknown option         */
      signal INVALID_OPTION
  end
 end
```

```
signal OK
 /* Error Routines                                        */

 NAME_MISS:
   say 'File name and file type must be specified'
   signal EXIT

 NAME_ERROR:
   say '"'fn ft'" is an incorrect file ID'
   signal EXIT

 NO_NUM_ERROR:
   say 'The value for NUMRECS has been omitted'
   signal EXIT

 INV_NUM_ERROR:
   say '"'num'" is an invalid positive number for NUMRECS'
   signal EXIT

 INVALID_OPTION:
   say '"'opt'" is an invalid option'
   signal EXIT

 OK:
 /* At this point, all of our flags and values have been  */
 /* set, and we are finally ready to process the file.

 . . .                              */
```

Figure 65. REXX Program Performing Its Own Syntax Checking

Processing MYCMD1 from an Assembler Program

The following two sample programs process MYCMD1. The first program, “Assembler Program Calling the Parsing Facility” on page 400, invokes the parsing facility using the PARSECMD macro. In this way, you do not need extra code to handle parsing. This program assumes it was invoked by CMSCALL. The second program, “Assembler Program Performing Its Own Syntax Checking” on page 401, includes code for parsing abbreviations, missing operands, and extra operands as well as code that issues error messages.

Assembler Program Calling the Parsing Facility

```
*********************************************************************
*
* ROUTINE:  SSORT
* FUNCTION: TAKE 2 STRINGS AND DISPLAY THEM IN EITHER ASCENDING
*           OR DESCENDING ORDER
* SYNTAX:   SSORT {ASCENDING|DESCENDING} STRING1 STRING2
* DLCS:     :CMD FFSSORT SSORT SSORT 5 :;
*             :OPR KWL(<ASCENDING 3><DESCENDING 4>) :;
*             :OPR FCN(STRING) :;
*             :OPR FCN(STRING) :;
*
*********************************************************************
SSORT    START
         USING *,12
         LR    12,15              ESTABLISH ADDRESSABILITY
         ST    14,R14SAVE         SAVE RETURN ADDRESS
*********************************************************************
* PARSE SSORT COMMAND
*********************************************************************
         LA    3,PARSLBL          GET ADDRESS OF PARSERCB STORAGE
         USING USERSAVE,13
         USING PARSERCB,3
         PARSECMD MF=(E,PARSLBL),UNIQID=UID,PLIST=(1),                *
               EPLIST=(0),ERROR=EXIT,CALLTYP=USECTYP
         DROP  13
         USING PVCENTRY,10
         L     10,PARPVCAD        GET PARSER VALIDATION CODE TABLE
         L     10,PVCNEXTA        POINT TO ENTRY OF ASCEND/DESCEND OPR
         L     9,PVCETOKA         GET ADDRESS OF ASCEND/DESCEND OPR
         L     10,PVCNEXTA        POINT TO ENTRY OF 1ST STRING
         L     5,PVCETOKA         GET ADDRESS OF 1ST STRING
         L     6,PVCETOKL         GET LENGTH OF 1ST STRING
         L     10,PVCNEXTA        POINT TO ENTRY OF 2ND STRING
         L     7,PVCETOKA         GET ADDRESS OF 2ND STRING
         L     8,PVCETOKL         GET LENGTH OF 2ND STRING
*********************************************************************
* DISPLAY STRING1 AND STRING2 IN EITHER ASCENDING OR DESCENDING ORDER
*********************************************************************
         CR    6,8                WHICH STRING HAS FEWER CHARS ?
         BH    COMP2              2ND STRING, TAKE BRANCH
         BCTR  6,0                DECREMENT FOR EXECUTE
         EX    6,COMPARGS         COMPARE STRINGS
         LA    6,1(,6)            INCREMENT BACK
         BNH   SMALL1             IF 1ST STRING GOES 1ST, BRANCH
         B     SMALL2             IF 2ND STRING GOES 1ST, BRANCH
COMP2    DS    0H
         BCTR  8,0                DECREMENT FOR EXECUTE
         EX    8,COMPARGS         COMPARE STRINGS
         LA    8,1(,8)            INCREMENT BACK
         BNL   SMALL2             IF 2ND STRING GOES 1ST, BRANCH
SMALL1   DS    0H
         CLI   0(9),C'D'          WANT TO SORT IN DESCENDING ORDER ?
         BE    TYPE21             YES, TYPE 2ND FOLLOWED BY 1ST
TYPE12   DS    0H
         WRTERM (5),(6)           WRITE OUT THE 1ST STRING
         WRTERM (7),(8)           WRITE OUT THE 2ND STRING
         B     GOODEXIT           EXIT WITH RC = 0
SMALL2   DS    0H
         CLI   0(9),C'D'          WANT TO SORT IN DESCENDING ORDER ?
         BE    TYPE12             YES, TYPE 1ST FOLLOWED BY 2ND
TYPE21   DS    0H
         WRTERM (7),(8)           WRITE OUT THE 2ND STRING
         WRTERM (5),(6)           WRITE OUT THE 1ST STRING
GOODEXIT DS    0H
         SR    15,15              ZERO OUT RC
EXIT     DS    0H
         L     14,R14SAVE         GET RETURN ADDRESS
         BR    14                 RETURN
PARSLBL  PARSECMD MF=L            GET INITIALIZED PARSERCB
UID      DC    CL16'FFSSORT'      UNIQUE ID FOR PARSECMD
R14SAVE  DS    A                  RETURN ADDRESS
COMPARGS CLC   0(*-*,5),0(7)      COMPARE STRINGS
         PARSERCB
         PVCENTRY
         USERSAVE
         END
```

Assembler Program Performing Its Own Syntax Checking

```
*********************************************************************
*
* ROUTINE:  LSORT
* FUNCTION: TAKE 2 STRINGS AND DISPLAY THEM IN EITHER ASCENDING
*           OR DESCENDING ORDER
* SYNTAX:   LSORT {ASCENDING|DESCENDING} STRING1 STRING2
* REQUIREMENTS: MUST GENMOD WITH SYSTEM OPTION
*
*********************************************************************
LSORT    START
         USING *,12
         LR    12,15              ESTABLISH ADDRESSABILITY
         ST    14,R14SAVE         SAVE RETURN ADDRESS
*********************************************************************
* PARSE LSORT COMMAND
*********************************************************************
         LR    11,0               GET EPLIST ADDRESS
         USING EPLIST,11
         L     9,EPLARGBG         GET ADDRESS OF 1ST ARG
         L     10,EPLARGND        GET END OF ARGS ADDRESS
         DROP  11
         SR    10,9               GET LENGTH OF ARGS FIELD
         LTR   10,10              DOES ARG1 EXIST ?
         BZ    MISSARG1           NO, ISSUE MESSAGE
         LA    1,0(10,9)          POINT PAST END OF ARGS FIELD
         LR    3,9                GET ADDRESS FOR EXECUTE
         BCTR  10,0               DECREMENT FOR EXECUTE
         EX    10,FINDEND         FIND END OF ARG 1
         LA    10,1(,10)          INCREMENT BACK
         SR    1,9                GET LENGTH OF ARG 1
         BZ    MISSARG1           IF LENGTH 0, MISSING ARG 1
         LR    11,1               SAVE LENGTH OF ARG 1
         BCTR  11,0               DECREMENT FOR EXECUTE
         EX    11,UPCASE          UPPERCASE ARG 1
         LA    11,1(,11)          INCREMENT BACK
TRYASC   DS    0H
         C     11,ASCMINL         ARG LENGTH LESS THAN MIN FOR ASCEND ?
         BL    TRYDESC            YES, TRY DESCENDING
         C     11,ASCMAXL         ARG LENGTH TOO BIG FOR ASCENDING ?
         BH    TRYDESC            YES, TRY DESCENDING
         BCTR  11,0               DECREMENT FOR EXECUTE
         EX    11,COMPASC         SEE IF ASCENDING WAS SPECIFIED
         LA    11,1(11)           INCREMENT BACK
         BE    GETSTRG1           IF ASCENDING, GET STRINGS
TRYDESC  DS    0H
         C     11,DESCMINL        ARG LENGTH LESS THAN MIN FOR DESCEND?
         BL    BADARG1            YES, ARG1 IS BAD
         C     11,DESCMAXL        ARG LENGTH TOO BIG FOR DESCENDING ?
         BH    BADARG1            YES, ARG1 IS BAD
         BCTR  11,0               DECREMENT FOR EXECUTE
         EX    11,COMPDESC        SEE IF ASCENDING WAS SPECIFIED
         LA    11,1(11)           INCREMENT BACK
         BNE   BADARG1            IF NOT DESCENDING, ARG IS BAD
GETSTRG1 DS    0H
         SR    10,11              ADJUST LENGTH OF ARGS FIELD
         BZ    MISSARG2           IF 0, MISSING ARG 2
         LA    4,0(11,9)          POINT PAST END OF ARG 1
         LR    1,4                GET FOR EXECUTE
         BCTR  10,0               DECREMENT FOR EXECUTE
         EX    10,FINDSTRT        FIND START OF STRING 1
         LA    10,1(,10)          INCREMENT BACK
         BZ    MISSARG2           ARG 2 MISSING, ISSUE MESSAGE
         LR    5,1                REMEMBER ADDRESS OF STRING1
         LR    2,5                GET ADDRESS OF STRING 1
         SR    2,4                - ADDRESS OF 1ST DEL AFTER ARG 1
         SR    10,2               ADJUST LENGTH OF ARGS FIELD
         LA    1,0(10,5)          POINT PAST END OF ARGS FIELD
         LR    3,5                GET ADDRESS OF STRING 1
         BCTR  10,0               DECREMENT FOR EXECUTE
         EX    10,FINDEND         FIND END OF STRING 1
         LA    10,1(,10)          INCREMENT BACK
         BZ    MISSARG3           NO DELIMS, MISSING STRING 2
         SR    1,5                GET LENGTH OF STRING 1
         BZ    MISSARG2           IF LENGTH 0, MISSING ARG 2
         LR    6,1                SAVE LENGTH OF STRING 1
GETSTRG2 DS    0H
         SR    10,6               ADJUST LENGTH OF ARGS FIELD
         BZ    MISSARG3           IF 0, MISSING ARG 3
         LA    4,0(6,5)           POINT PAST END OF STRING 1
```

```
         LR    1,4                GET FOR EXECUTE
         BCTR  10,0               DECREMENT FOR EXECUTE
         EX    10,FINDSTRT        FIND START OF STRING 2
         LA    10,1(,10)          INCREMENT BACK
         BZ    MISSARG3           ARG 3 MISSING, ISSUE MESSAGE
         LR    7,1                REMEMBER ADDRESS OF STRING 2
         LR    2,7                GET ADDRESS OF STRING 2
         SR    2,4                - ADDRESS OF 1ST DEL AFTER ARG 2
         SR    10,2               ADJUST LENGTH OF ARGS FIELD
         LA    1,0(10,5)          POINT PAST END OF ARGS FIELD
         LR    3,7                GET ADDRESS OF STRING 2
         BCTR  10,0               DECREMENT FOR EXECUTE
         EX    10,FINDEND         FIND END OF STRING 2
         LA    10,1(,10)          INCREMENT BACK
         BZ    GETLEN             NO DELIMS, USE LENGTH OF ARGS FIELD
         SR    1,7                GET LENGTH OF STRING 2
         BZ    MISSARG3           IF LENGTH 0, MISSING ARG 3
         LR    8,1                SAVE LENGTH OF STRING 2
         B     CHKEXTRA           SEE IF EXTRA OPERANDS SPECIFIED
GETLEN   DS    0H
         LR    8,10               USE LENGTH LEFT OF ARGS FIELD
CHKEXTRA DS    0H
         SR    10,8               ADJUST LENGTH OF ARGS FIELD
         BZ    SORTSTRG           IF 0, ALL OK
         LA    4,0(8,7)           POINT PAST END OF STRING 2
         LR    1,4                GET FOR EXECUTE
         BCTR  10,0               DECREMENT FOR EXECUTE
         EX    10,FINDSTRT        FIND EXTRA OPERANDS
         LA    10,1(,10)          INCREMENT BACK
         BZ    SORTSTRG           NO EXTRA OPERANDS, ALL OK
         LR    2,1                GET ADDRESS OF EXTRA OPERANDS
         SR    2,4                - ADDRESS OF 1ST DEL AFTER ARG 3
         SR    10,2               GET LENGTH OF EXTRA OPERANDS
         LR    2,1                GET ADDRESS OF EXTRA OPERANDS
         B     EXTRAOP            EXTRA OPERANDS, ISSUE MESSAGE
*********************************************************************
* DISPLAY STRING1 AND STRING2 IN EITHER ASCENDING OR DESCENDING ORDER
*********************************************************************
SORTSTRG DS    0H
         CR    6,8                WHICH STRING HAS FEWER CHARS ?
         BH    COMP2              2ND STRING, TAKE BRANCH
         BCTR  6,0                DECREMENT FOR EXECUTE
         EX    6,COMPARGS         COMPARE STRINGS
         LA    6,1(,6)            INCREMENT BACK
         BNH   SMALL1             IF 1ST STRING GOES 1ST, BRANCH
         B     SMALL2             IF 2ND STRING GOES 1ST, BRANCH
COMP2    DS    0H
         BCTR  8,0                DECREMENT FOR EXECUTE
         EX    8,COMPARGS         COMPARE STRINGS
         LA    8,1(,8)            INCREMENT BACK
         BNL   SMALL2             IF 2ND STRING GOES 1ST, BRANCH
SMALL1   DS    0H
         CLI   0(9),C'D'          WANT TO SORT IN DESCENDING ORDER ?
         BE    TYPE21             YES, TYPE 2ND FOLLOWED BY 1ST
TYPE12   DS    0H
         WRTERM (5),(6)           WRITE OUT THE 1ST STRING
         WRTERM (7),(8)           WRITE OUT THE 2ND STRING
         B     GOODEXIT           EXIT WITH RC = 0
SMALL2   DS    0H
         CLI   0(9),C'D'          WANT TO SORT IN DESCENDING ORDER ?
         BE    TYPE12             YES, TYPE 1ST FOLLOWED BY 2ND
TYPE21   DS    0H
         WRTERM (7),(8)           WRITE OUT THE 2ND STRING
         WRTERM (5),(6)           WRITE OUT THE 1ST STRING
GOODEXIT DS    0H
         SR    15,15              ZERO OUT RC
         B     EXIT               EXIT
EXTRAOP  DS    0H
         APPLMSG NUM=070,CSECT=SSO,SUB=(CHARA,((2),(10)))
         B     BADRC              EXIT WITH RC = 24
BADARG1  DS    0H
         APPLMSG NUM=388,CSECT=SSO,SUB=(CHARA,((9),(11)))
         B     BADRC              EXIT WITH RC = 24
MISSARG1 DS    0H
MISSARG2 DS    0H
MISSARG3 DS    0H
         APPLMSG NUM=386,CSECT=SSO
BADRC    DS    0H
         LA    15,24              GET BAD RC
EXIT     DS    0H
         L     14,R14SAVE         GET RETURN ADDRESS
         BR    14                 RETURN
```

```
FINDEND  TRT   0(*-*,3),DELIMS    FIND NEXT DELIMITER
FINDSTRT TRT   0(*-*,4),NONDELIM  FIND NEXT NONDELIMITER
UPCASE   OC    0(*-*,9),BLANKS    UPPERCASE ARGUMENT
COMPASC  CLC   0(*-*,9),ASCEND    CHECK FOR ASCENDING OPERAND
COMPDESC CLC   0(*-*,9),DESCEND   CHECK FOR DESCENDING OPERAND
COMPARGS CLC   0(*-*,5),0(7)      COMPARE STRINGS
         DS    0F
BLANKS   DC    10C' '             BLANKS FOR UPPERCASING OPERAND
DELIMS   DC    64X'00',C' ',12X'00',C'(',178X'00'
NONDELIM DC    64X'FF',X'00',12X'FF',X'FF',178X'FF'
ASCEND   DC    C'ASCENDING'
ASCMINL  DC    F'3'
ASCMAXL  DC    F'9'
DESCEND  DC    C'DESCENDING'
DESCMINL DC    F'4'
DESCMAXL DC    F'10'
R14SAVE  DS    A                  RETURN ADDRESS
         DSECT
EPLIST   DS    0H
EPLCMD   DS    A
EPLARGBG DS    A
EPLARGND DS    A
         END
```

Creating and Distributing Your Own CMS Commands

Using DLCS

If you give users commands that call the parsing facility, put the syntax in a DLCS file that has a unique application identifier. In this way, users who receive your commands and syntax do not have to merge the syntax definition with their DMS tables.

See the example in Figure 62 on page 395. In this example, you could change the application identifier on the DLCS statement to your initials. For example, if your initials are AGW, the DLCS statement looks like:

```
:DLCS AGW USER AMENG :;
```

Then, enter the following PARSECMD command from a REXX program:

```
'PARSECMD MMYCMD1 (APPLID AGW'
```

To make it even easier for other users, you can automatically load and drop the table from storage by issuing the following SET LANGUAGE command, just before the PARSECMD command, from your REXX program:

```
'SET LANGUAGE (ADD AGW USER'
```

Then enter the following SET LANGUAGE command before exiting:

```
'SET LANGUAGE (DELETE AGW USER'
```

By doing this, all parsing is hidden, and users do not have to enter the SET LANGUAGE command.

Defining Translations, Synonyms, and Abbreviations

If users want to translate the command or the keywords of the command into their own national language, they have to edit the DLCS table you send them to translate the parameters. However, to translate commands or keywords into a language other than the default language, the other language must exist on your system.

If users want to define a translation for the command name, they can just add an entry in their DMS table. For example, if the user's translation for your command name is ‘FRIENDCMD’, the DMS table entry is:

```
:CMD ' ' MYCMD1 FRIENDCMD 5 :;
```

If users just want to abbreviate your command name, they can add an entry in their own DMS user tables that defines your command with the blank unique id:

```
:CMD ' ' MYCMD1 MYCMD1 3:;
```

To abbreviate the command name and define a synonym, such as ‘MC’, they can add:

```
:CMD ' ' MYCMD1 MYCMD1 3:;
  :SYN MC 2:;
```

After users define translations, synonyms and abbreviations, they must do a run GENCMD against the files they have changed. To enable user additions to the DMS table they must issue SET LANG (ADD DMS USER.

Note: This application does not support DBCS tokens unless there is already a system table available for the application.

Defining HELP Files

You can also create HELP text files for your own commands. By specifying the appropriate HELP command, you can display information about the commands you created. See the *z/VM: CMS User's Guide* for details on creating your own HELP files.

Chapter 26. Using Message Repository Files

This chapter describes how to:

- Create and use message repositories
- Use substitution in your message repositories
- Use dictionary substitution in your message repositories
- Create your own CMS messages
- Create your own HELP files
- Make your messages available to others.

When you write a program and you want to display error messages, you can put message text directly in your application. However, if you have many messages, your programs can become cluttered. Instead of coding message text directly in an application, you can store all your message texts in one file called a ***repository***. Then, to display a message, your application can retrieve the message text from this repository.

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

- Message text does not clutter your application.
- You can access the same message from many applications without having to specify the message text each time.
- You can have your messages translated into other languages. You can then have your messages in the language you want.
- You can create your own message file for whatever application you want to run, including CMS.

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

Note:

1. For languages other than English available on your system, the file name of the CMS message repository is different. For more information, see the *z/VM: Installation Guide*.
2. If an application supplies a system message repository (*xxx*MES*cc* TEXT, generated from a system DLCS file), application users can override and customize the application messages by creating user message repository files. However, if an application supplies a user message repository (*xxx*UME*cc* TEXT, generated from a user DLCS file), application users can change the application messages only by modifying the DLCS source file and generating a new TEXT file.
3. Application and user message repositories can be created only for the languages supported by z/VM. (See *z/VM: General Information* for the list of supported languages.) User message repositories are made available by using the CMS SET LANGUAGE command, and there must be a supported corresponding system language repository to which to add the user repositories.

Creating and Using Message Repositories

To create and use your own message repository file, follow these steps:

1. Create a message repository file using XEDIT.
2. Check the accuracy of the message repository notation and convert the repository into machine readable form.
3. Make the repository file available for the language you are working with.
4. Access the messages from your application.

Step 1. Creating a Message File

To create your own message repository, use XEDIT. The file name of your repository must be *xxx*UME*cc*, where:

- *xxx* is an application identifier. It must be three alphanumeric characters, and the first character must be alphabetic (for example, DMS, HCP, OFS, AGW, DKK, and so on).
- UME refers to User MEssage repository.
- *cc* is a 1- or 2-character country code that identifies the language in which you are working. The country code is the first field of each record in the VMFNLS LANGLIST file on the S-disk. A country code is not used for American English. For more information about the VMFNLS LANGLIST file, see *z/VM: VMSES/E Introduction and Reference*.

The message repository can have any valid CMS file type not reserved for some other function. For a list of reserved CMS file types, see *z/VM: CMS User's Guide*. The convention is to use a file type of REPOS.

Your message repository should contain the following items:

- Comment records
- A control line
- Message records.

Note: The file must be in fixed-record format, and have a maximum LRECL of 80.

Commenting Your Message Repository

Your message repository file should contain comment records. These must start with an asterisk in column 1.

```
* This is an example of a comment line
```

Comment lines can go anywhere in a message repository file and should describe what is in the file.

Creating a Control Line

The first noncomment record in your external repository must be a control line. This control line contains two things:

1. A character that specifies substitutions. This must be the first nonblank character on the control line.
2. A number that specifies how many message number digits (3 or 4) you want to display in the message header. This must be the second nonconsecutive, nonblank character on the control line. If the number is 4 digits you will get 4 digits even if you specify only 3.

For example, the following control line specifies & as the substitution character and uses three digits to display the message number:

```
& 3
```

Creating Message Records

Each message record in a repository file contains five fields. When you create your file using XEDIT, every message record must be in the following format:

```
=====        .
=====        .
=====        .
===== NNNNFFLLS ------------------------ text ------------------------------
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7...
=====        .
=====        .
=====        .
```

Figure 66. Message Record Format

NNNN
is the message number, in columns 1-4. You must use a 4 digit message number in a message repository. The CMS system repository uses message numbers 0001 through 7999.

You do not have to place messages in sequence by message number in a CMS (or CMS application) user repository. However, message numbers do have to be in sequential order in a CP repository.

FF
is the message format, in columns 5-6. This field is for a message that can be in several versions. If a message has just one format, you do not need to type anything—the format field defaults to “01”. You cannot use “00” as a format number.

LL
is the line number of the message, in columns 7-8. Use this field to define text for a single message format that may span more than one line. Messages that spread across more than one line must have sequential, consecutive line numbers.

If a single format of a message has just one line, you do not need to type a line number, the line number defaults to “01”. You cannot use “00” as a line number.

S
is the severity code, in column 9. Your severity codes could be one of the following:

Code
Message Type

E
Error

I
Information

R
Response

S
Severe

T
Terminal

W
Warning

You can define other severity codes.

text
is the message text, starting in column 11. You can specify up to 62 characters of message text on one line. If the text for a single message is longer than 62 characters, you must put the message text on more than one line and specify the same message number for each.

If you want multiline message text displayed on the screen in one continuous line (wrapped around), the message number (NNNN), the message format (FF), and the line number (LL) must be identical for each line.

If you want multiline message text displayed on the screen in more than one line, you must make the first line number 01 and line numbers after that 02, 03, etc.

When a single line message is created from a multiline definition, all trailing blanks but one are stripped from the input lines and the lines are concatenated together. If you want to build a line with many blanks, you can do this two ways:

- If you want more than one blank between two items, split the items onto two lines. Insert as many blanks as you need before the item on the second line. However, you must also specify NOCOMPRESS on the XMITMSG command or COMP=NO on the APPLMSG macro.
- If you want more than one blank between words or if you want to maintain alignment of fields, you can use a substitution variable that will have a null or omitted value. See the sample repository in Figure 70 on page 412 and sample program in Figure 71 on page 412.

Example of a Message Repository

The following example shows an external repository file, DIAUME REPOS. You can view, edit, and update this message repository.

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


00000 * * * Top of File * * *
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7...
00001 *
00002 * This is an example of a message repository file for a small
00003 * programming application.
00004 * This was created via XEDIT.
00005 * You can code a file similar to this for your own application.
00006 *
00007 & 3 Specifies the substitution character and the number of digits
00008 00050101E Invalid syntax; please reissue command.
00009 00150101R Enter the number of copies you want:
00010 00250101I Function has completed
00011 00250201I Subroutine has completed
00012 01000101R Your program has just halted at label ABCD.
00013 01000102R You can quit the program by entering 'Q', or
00014 01000103R press the ENTER key to continue.
00015 * * * End of File * * *
```

Figure 67. Sample Repository - DIAUME REPOS

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

Line(s)
Explanation

1 - 6
Comment lines.

7
The control line.

The first nonblank character on this line (&) specifies the substitution character for messages. (See “Using Substitution in a Message Repository” on page 410.) The second nonblank character specifies that you want to display three message number digits (the default).

Note: If a message number is greater than 999, then 4 digits are displayed regardless of the control line number.

8
The first message in the repository is number 0005. The message results from a user error, so the severity (column 9) is “E”.

9
The second message is number 0015. The message is requesting input from a user, so the severity (column 9) is “R”.

10 - 11
The third message is number 0025. This message has two formats: depending on the error, either `Function has completed` (format 01) or `Subroutine has completed` (format 02) is displayed. These messages give the user information, so the severity (column 9) is "I".

12 - 14
The fourth message is number 0100. This message has only one format, but it spreads across three lines of the repository. Columns (7-8) show the line numbers of this message. The message requests input from a user, so the severity (column 9) is "R".

Step 2. Checking and Compiling Message Repository File

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

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

You can also use the GENMSG command without the NOOBJECT option to check for errors and compile the message repository file. However, it is recommended that you use the GENMSG command with the NOOBJECT option the first time you check for syntax errors. This eliminates the process of compiling the message repository file that may have errors.

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

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

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

Example 1: Enter the following GENMSG command to check for errors in your message repository file called DIAUME REPOS:

```
GENMSG DIAUME REPOS A DIA (NOOBJECT
```

DIAUME REPOS A is the file ID of your message repository you created. DIA is the *applid*, which is the operand used to identify your application. This application identifier must be three characters long and must correspond to the first three letters of the repository file name. Be sure to record the application identifier you choose. You will need to reference it when you access your messages by the XMITMSG command or APPLMSG macro.

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

```
GENMSG DIAUME REPOS A DIA
```

Example 2: Enter the following GENMSG command to check for errors in your message repository file called DIAUME REPOS and to compile DIAUME REPOS:

```
GENMSG DIAUME REPOS A DIA
```

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

Step 3. Making Message File Available

Once you compile the message file, you must make the message file active for the language you are working in. You accomplish this with the SET LANGUAGE command. For example, DIAUME REPOS can be made available by entering:

```
SET LANGUAGE (ADD DIA USER
```

You may also want to create your own CMS message repository and make it available to an application. See "Creating Your Own CMS Messages" on page 414 for an explanation of how to do this.

Note: Each time you make a change to the message repository, you must issue the GENMSG and SET LANGUAGE commands.

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

Step 4. Accessing Messages

To access messages from a repository, use the XMITMSG command. For example, to display this CMS message, found in DIAUME REPOS, from your application:

```
Subroutine has completed
```

you could use the following XMITMSG command from the CMS command line:

```
XMITMSG 25 (APPLID DIA FOR 2
```

See the *z/VM: CMS Commands and Utilities Reference* for a complete description of the XMITMSG command.

Note: You can also access message repositories from assembler applications using the APPLMSG macro. See the *z/VM: CMS Application Development Guide for Assembler* and *z/VM: CMS Macros and Functions Reference* for information on how to use the APPLMSG macro.

Using Substitution in a Message Repository

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

```
Invalid option 'GO'
```

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

```
Invalid option 'FILE'
Invalid option 'RUN'
Invalid option 'STOP'
```

You do not need four separate messages in your repository. Instead, you can create a single message text that contains a substitution variable and then substitute different information using this substitution variable. For example, your repository could look like Figure 68 on page 411.

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


===== * * * Top of File * * *
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7...
===== *
===== * This is an example of a message repository file that
===== * uses simple substitution.
===== *
===== & 3 Line specifies the substitution character + no. of digits
===== 02000101  Invalid option &1
===== * * * End of File * * *
```

Figure 68. Sample Repository - SAPUME REPOS

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

Here are some rules about substitutions:

- A substitution can be a single word, a phrase, or an entire sentence.
- A substitution can go anywhere within a message.
- You can have more than one substitution per message.

The data that replaces the &1, &2, and so on, can come from the program itself (for example, an operand on the XMITMSG call) or from a dictionary. The following is an example of an XMITMSG call to the repository in Figure 68 on page 411.

```
/*  */
sub = "'FILE'"
'XMITMSG 200 SUB (APPLID SAP NOHEADER'
exit
```

Figure 69. Sample Code Accessing SAPUME EXEC

The result of the previous XMITMSG call is:

```
Invalid option ‘FILE’
```

For information about how to specify substitutions on the XMITMSG command (literal substitutions and dictionary substitutions are specified differently), see *z/VM: CMS Commands and Utilities Reference*.

Example of Using Substitution in a Message Repository

Substitution can also be used to build messages with many blanks or to maintain alignment. The repository in Figure 70 on page 412 illustrates the use of a substitution character in this manner.

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


00000 * * * Top of File * * *
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7...
00001 *
00002 * This is an example of a message repository file made via XEDIT
00003 * and maintain alignment of heading.
00004 * You can code a file similar to this for your application.
00005 *
00006 & 3 Line specifies the substitution character + no. of digits.
00007 00990101  |...+....1....+....2....+....3....+....4....+....5...
00008 00990101  .+....6....+....7....+....8
00009 01000101  *                                                 &2
00010 01000101                              *
00011 02000101  Header One         &2
00012 02000101  Header Two         &2
00013 02000101  Header Three       &2
00014 02000101  Header Four        &2
00015 * * * End of File * * *
```

Figure 70. Sample Repository - RUBUME REPOS

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

Line(s)
Explanation

1 - 5
Comment lines.

6
The control line.

An ampersand (&) is the substitution character, and you want to display 3 message number digits.

7 - 8
The first message is number 0099. This message has only one format and will be displayed as one line.

9 - 10
The second message is number 0100. This message has only one format and will be displayed as one line. The substitution character, &2, has a value of null and maintains alignment.

11 - 14
The third message is number 0200. This message has only one format and will be displayed as one line. The substitution character, &2, has a value of null and maintains alignment.

Before accessing the messages in RUBUME REPOS, you must compile and activate the message file. In this example, when you issue the GENMSG command to compile the message file, you should specify the MARGIN 63 option. This indicates that the message should be taken from columns 1-63 in the input deck (RUBUME REPOS). Then, issue the SET LANGUAGE command to activate the message file. Finally, to access messages in RUBUME REPOS, use the following exec:

```
/*                                                  */
dummy = ''
'XMITMSG  99       ( APPLID RUB NOHEADER'
'XMITMSG 100 DUMMY ( APPLID RUB NOHEADER NOCOMPRESS'
'XMITMSG 200 DUMMY ( APPLID RUB NOHEADER NOCOMPRESS'
exit
```

Figure 71. Sample Program - RUBUME EXEC

In the second call to XMITMSG:

200
specifies that message number 200 will be displayed.

DUMMY
is a substitution variable. In this case, a null character is substituted, which is necessary when NOCOMPRESS is specified. Otherwise, the substitution indicators, &1, &2, ..., specified in the message repository, appear in the displayed message.

APPLID RUB
specifies RUBUME REPOS as the message repository that issues the message.

NOHEADER
specifies that the message header will not be displayed on the terminal.

NOCOMPRESS
specifies that the blanks will not be removed.

The output from this exec will look like this:

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

Using Dictionary Substitution in a Message Repository

Each dictionary record contains a 4 digit message identifier and dictionary text. These records are stored in the repository file along with the messages. You can make the first digit of the message identifier a certain number (8, for example) that shows the item is a dictionary item.

Example of Using Dictionary Substitution in a Message Repository

The following is a sample message repository that contains a two-item dictionary:

```
 OPLUME   REPOS    A1  F 80  Trunc=80 Size=9 Line=1 Col=1 Alt=0


00000 * * * Top of File * * *
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7...
00001 *
00002 * This is an example of a message repository file made with XEDIT
00003 * You can code a file similar to this for your application.
00004 *
00005 & 4 Line specifies the substitution character + no. of digits
00006 00070101E Invalid option '&1'
00007 00170101I You have invoked the &1
00008 80010101  compiler
00009 80020101  assembler
00010 * * * End of File * * *
```

Figure 72. Sample Repository - OPLUME REPOS

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

Line(s)
Explanation

1 - 4
Comment lines.

5
The control line.

An ampersand (&) is the substitution character, and you want to display 4 message number digits.

6
The first message in the repository, number 0007. When the message is to be displayed, you have to specify what information is to be substituted in place of the &1.

7
The second message, number 0017. When the message is to be displayed, you have to specify what information is to be substituted in place of the &1.

8

A dictionary item, number 8001. If you want to call message 0017 substituting “compiler” into &1, use the following XMITMSG command from the command line:

```
XMITMSG 17 8001 (APPLID OPL
```

The following message is displayed:

```
OPL???0017I You have invoked the compiler
```

(The message header also includes the application ID 0PL and ??? is the default for the caller. See *z/VM: CMS Commands and Utilities Reference* for more information on XMITMSG.)

9

A second dictionary item, numbered 8002. If you want to call message 0017 substituting “assembler” into &2, use the following XMITMSG command:

```
XMITMSG 17 8002 (APPLID OPL
```

The following message is displayed:

```
OPL???0017I You have invoked the assembler
```

Creating Your Own CMS Messages

As noted earlier, the CMS system repository uses numbers 001-7999 for CMS messages. It also uses 8000-9999 for dictionary items and unnumbered responses. You can view this file using XEDIT, and you could print off a copy for you to use as reference.

You can create your own CMS messages and put them in a separate user CMS repository. A user CMS repository can contain message numbers that are additions to existing CMS messages or duplicate message numbers. If your CMS repository contains message numbers that duplicate existing CMS messages, your version overrides the system version.

To build your own CMS message repository, follow these steps:

1. To create a CMS message file of your own messages, use the XEDIT command. The file name must be DMSUME.
2. To compile your message file, use the GENMSG command:

   ```
   genmsg dmsume ft fm dms
   ```

   Your compiled message will have a file name of DMSUME, a file type of TEXT, and an application identifier of DMS.

   See the *z/VM: CMS Commands and Utilities Reference* for a description of the GENMSG command.
3. To activate your user CMS repository along with the CMS system repository, DMSMES REPOS, use the SET LANGUAGE command:

   ```
   set language (add dms user
   ```

   See the *z/VM: CMS Commands and Utilities Reference* for a description of the SET LANGUAGE command.

For example, suppose you wanted to add your own informational message that says:

```
The command you issued takes five minutes to complete.
```

You can enter this message in your CMS repository in two different ways:

1. Using your own unique message number.

   You can look in the CMS system repository, DMSMES REPOS, for the message numbers that are not used. Then, you can put these available message numbers in your own message repository, DMSUME

REPOS. For example, suppose number 1000 is available. You can add the following line to DMSUME REPOS:

```
⋮
10000101I The command you issued takes five minutes to complete.
⋮
```

Then, compile DMSUME REPOS using the following GENMSG command:

```
genmsg dmsume repos a dms
```

Make the file active using the following SET LANGUAGE command:

```
set language (add dms user
```

And, access the message using the following XMITMSG command from the CMS command line:

```
xmitmsg 1000 (disp for 1 applid dms comp noheader
```

Note: Be aware that the CMS system repository may be updated in the future to include a message number you used in your message repository. If this occurs and you access this message number, you do not see the CMS system message, you see your own version of the message.

2. Using an existing CMS message number.

   The CMS system repository contains a message:

   ```
   00460101E  No library name specified
   ```

   Suppose you enter the message:

   ```
   00460101E The command you issued takes five minutes to complete.
   ```

   compile the repository file, and then make it active. When you access message 00460101E, you do not see the CMS system message, you see your own version of the message.

If any of your own messages require dictionary substitutions, you should note this restriction: You must include dictionary items for your messages in your own message repository—you cannot access a message from one repository, using dictionary items from another.

Creating Your Own HELP Files

You can also create HELP text files for your own messages. The HELP files contain explanatory information about these messages. By specifying the appropriate HELP command, you can display information about the messages you created. See the *z/VM: CMS User's Guide* for details on creating your own HELP files.

Making Your Messages Available to Others

When you make your own repository and issue a SET LANGUAGE command, that repository is available only to your virtual machine. However, you may want to allow other users on your system to access your messages. You can accomplish this by either of the following two methods:

1. Have other users link to your disk. They must then issue a SET LANGUAGE command for their virtual machine.
2. Have your message file placed in shared storage (a saved segment) so all users can access it.

Loading a User Message Repository into a CMS Logical Saved Segment

The following steps outline the procedure for loading a user message repository into a CMS logical saved segment. For additional information about defining and building a logical saved segment, see *z/VM: CP Planning and Administration*.

In this example, the repository contains American English messages for an application whose application ID is LCL. The name of the compiled message repository (the output of the GENMSG command) is LCLUME TEXT.

1. Create a LANGMERG control file for the application (or update the existing control file for the application) to identify the compiled message repository and any other language-related files for the application that are to be combined into a single text file. The message repository is identified on a LANGUAGE record in the control file. In this example, a new control file named LCLAMENG LANGMCTL is created. It contains the following records:

   ```
   DISK 19D
   ETMODE OFF
   MESSAGE LCLUME (VMCTL DMSVM
   ```

   For information about the content and format of the LANGMERG control file, see the description of the LANGMERG command in the *z/VM: CMS Commands and Utilities Reference*.

2. Use the LANGMERG command to combine the language information into a single text file:

   ```
   langmerg ameng lcl
   ```

   LANGMERG generates a file named LCLNLS TXTAMENG.

3. Define a CP saved segment to use as the physical saved segment that holds the logical saved segment where you intend to load the user message repository. (If you plan to define the logical saved segment in an existing physical saved segment, you can skip this step.) Use the DEFSEG command to define either a member saved segment within a segment space or a stand-alone discontiguous saved segment (DCSS). In this example, a DCSS named LCL is created, of type SR (shared read-only access), located in pages 600-6FF:

   ```
   defseg lcl 600-6ff sr
   ```

   Note:

   a. For more information about the DEFSEG command, see the *z/VM: CP Commands and Utilities Reference*.

   b. To define a CP saved segment you need CP class E command privileges. You may have to contact your system support personnel.

   c. If you prefer, you can use VMSES/E (the VMFSGMAP and VMFBLD commands) to define and build your saved segments. In that case, omit this step and steps "5" on page 416 and "7" on page 417. For more information about using VMSES/E to define and build saved segments, see *z/VM: CP Planning and Administration*.

4. If you want to add the message repository to an existing logical saved segment, edit the logical segment definition file and add a LANGUAGE record that identifies the language file; otherwise, create a new logical segment definition file. When the segments are built, the file name of the logical segment definition file becomes the name of the logical saved segment. In this example, a logical segment definition file named LCLAMENG LSEG is created for a new logical saved segment named LCLAMENG. It contains the following record:

   ```
   LANGUAGE LCL AMENG
   ```

5. If you want to add the logical saved segment to an existing physical saved segment, edit the physical segment definition file and add a logical segment (LSEGMENT) record that identifies the logical segment definition file; otherwise, create a new physical segment definition file. The file name of the physical segment definition file must be the name of the DCSS or member saved segment you are using as the physical saved segment. In this example, a physical segment definition file named LCL PSEG is created for a new physical saved segment named LCL. It contains the following record:

   ```
   LSEGMENT LCLAMENG LSEG
   ```

6. The system segment identification file, usually named SYSTEM SEGID, associates each logical saved segment with its physical saved segment. It must reside on the CMS system disk. This file is updated

or created by the SEGGEN command. Because the file cannot be updated or created on the CMS system disk, you must make sure that a copy of the version you want to update is located on a read/write disk.

7. Use the SEGGEN command to build (or rebuild) the physical and logical saved segments:

   ```
   seggen lcl pseg a system segid a (map gen
   ```

   For information about using the SEGGEN command, see the *z/VM: CMS Commands and Utilities Reference*.
8. Copy the SYSTEM SEGID file to the system disk and resave CMS. There are special instructions for doing this, and some of the steps can be done only by authorized user IDs. You may have to contact your system support personnel. For more information, see the description of the SEGGEN command in the *z/VM: CMS Commands and Utilities Reference* or the chapter on planning and defining logical saved segments in *z/VM: CP Planning and Administration*.
9. Users can then use the SEGMENT LOAD command to load the logical saved segment into their virtual machines:

   ```
   segment load lclameng
   ```

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

Chapter 27. Using Saved Segments

This chapter describes:

- What physical and logical saved segments are.
- How to reserve, load, purge, release, and assign a saved segment using the SEGMENT command.
- How to display information about saved segments and segment storage spaces using the QUERY SEGMENT command.

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. 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:

- Several 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 other 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: CP Planning and Administration*.

Using the SEGMENT Command

You can use the SEGMENT command to:

- Reserve storage space for a saved segment
- Load the saved segment into storage
- Purge the 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.

Loading Saved Segments

The SEGMENT LOAD command loads a saved segment into storage. SEGMENT LOAD reserves virtual storage for the saved segment (if a storage space is not already reserved), then loads the saved segment into storage.

Example 1: To load a saved segment named MYSEG that (a) survives abend processing and (b) can be shared by any user, enter:

```
segment load myseg (system share
```

Note that SHARE is the default value and can be omitted.

Example 2: To load a saved segment named MYSEG that does not survive abend processing and cannot be shared by other users, enter:

```
segment load myseg (noshare user
```

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 within the same physical saved segment, you must specify (or default to) the same attribute.

For example, suppose a physical saved segment USERSEG contains logical saved segment MYSEG. If you specify the NOSHARE option when you load MYSEG, you must specify NOSHARE when you load any other logical saved segments in USERSEG. If the SHARE or NOSHARE option on the SEGMENT LOAD command does not match the SHARE attribute of the physical saved segment, the saved segment is ***not*** loaded and you receive a return code of 36.

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 command. 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 space for and loads the entire segment space.

4. If the requested saved segment is none of the above, the command returns a return code of 44.

CMS uses this same search order to 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, CSLs 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. Once 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.

Purging Saved Segments from Your Virtual Machine

The SEGMENT PURGE command purges a saved segment from a segment storage space.

For example, to purge MYSEG, enter:

```
segment purge myseg
```

Purging a logical saved segment removes the objects that it contains from use by CMS. If this logical saved segment is the only loaded or reserved saved segment within the physical saved segment, 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 storage area is also released unless it was explicitly reserved using the SEGMENT RESERVE command.

When you use the SEGMENT PURGE command to purge a saved segment, the saved segment must have been loaded using the SEGMENT LOAD command. 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 the SEGMENT PURGE command and the DIAGNOSE code X'64' PURGESYS function interchangeably.

To locate the saved segment you want to purge, SEGMENT PURGE uses the search order described under “How CMS Locates Saved Segments” on page 420.

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
```

Displaying Information about Saved Segments

The QUERY SEGMENT command displays information about the saved segments attached to a virtual machine and the storage spaces that contain or are reserved for saved segments. For example,

```
query segment myseg
```

might return a response similar to the following:

```
Space    Name     Location  Length    Loaded  Attribute
PSEG2    MYSEG    02240000  00380000  NO      SYSTEM
```

If MYSEG is a logical saved segment, then PSEG2 is the name of the physical saved segment.

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

```
query segment *
```

In response, CMS returns something similar to the following:

```
Space     Name      Location  Length    Loaded   Attribute
NLSUCENG  NLSUCENG  00DA0000  00060000  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.

The CONTENTS option displays the contents of a logical saved segment. 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
```

The ASSIGN option lists the logical saved segment name and the physical segment to which it is currently assigned, as follows:

```
query segment lseg1 assign
```

The response is in the following form:

```
Lsegname  Assignment
LSEG1     PSEG1
```

The SPACE option displays information about segment storage spaces. To display information about all 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 SEGMENT and QUERY SEGMENT commands, see the *z/VM: CMS Commands and Utilities Reference*.

Chapter 28. Using DB2 Server for VM

This chapter will cover the following topics:

- How DB2® Server for VM organizes the data.
- Description of the commands and how you use them
- How to create, query, and manipulate tables
- How to create views of a table
- How to preprocess your SQL application
- Description of Interactive SQL (ISQL) and Query Management Facility (QMF*).

A ***database*** is a centrally controlled, integrated collection of data. A database system controls the storing and retrieval of data. Database systems are useful because they can be used to:

- Reduce redundancy
- Avoid inconsistencies
- Share data among many users
- Enforce data processing standards
- Apply and maintain data integrity and security
- Resolve conflicting application requirements.

DB2 Server for VM is a relational database management system available for CMS users. DB2 Server for VM simplifies data handling by offering facilities for querying and manipulating data and writing reports. It also contains data recovery and data security measures.

This chapter provides a general introduction on how to use the Structured Query Language (SQL) in high-level language applications to access data stored in DB2 Server for VM tables. It is not intended to be a complete description of the use of DB2 Server for VM. You can use SQL in a program written in assembler, C, COBOL, FORTRAN, PL/I, or REXX.

Note: Refer to the appropriate SQL publications for additional information.

How SQL Handles Data

In DB2 Server for VM, the data is addressed by content, rather than by its location or organization in storage. It takes the form of tables in row and column format. DB2 Server for VM also keeps catalogs that serve as an integrated data dictionary and directory. These catalogs always reflect the current status of the database and are automatically updated.

Data is defined and accessed in terms of tables and operations on tables. A table is defined to DB2 Server for VM by identifying the ***columns*** in the table and their characteristics. All values in a column have the same characteristics. A table ***row*** is the smallest unit of insertion and deletion in DB2 Server for VM. An insert operation adds one or more rows to a table. A delete operation removes one or more rows from a table. The smallest unit of data update in DB2 Server for VM is the ***field***, which is the point where a specific row and column meet. A field contains a single ***data item***.

You can do the following table operations:

- Create or delete tables.
- Retrieve data by table, row, or field.
- Update, insert, or delete data.
- Add new columns to a table.
- Copy data from one table into another.
- Perform utility operations, such as bulk data loading, data reorganization, and printing.

DB2 Server for VM can also store ***indexes*** to particular columns in a table. You do not need indexes to access stored data, but they help DB2 Server for VM optimize its performance. When you request an index, DB2 Server for VM creates and maintains it. When you write a program to access data, you do not refer to the index explicitly, but DB2 Server for VM decides which index to use.

DB2 Server for VM can also store ***view definitions***. A view is a logical or virtual table derived from one or more tables. It is like a stored table with rows and columns. You can use views as if they were tables. However, some operations are not valid on views. Others are restricted, depending on how the view was defined. You can use views mainly to simplify data retrieval commands and to limit access to data or its manipulation.

Using SQL, you specify only the results you want. When you reference the data, you do not specify data paths, access methods, or the organization of the physical file.

SQL Commands

An SQL command contains a verb with one or more optional clauses, language keywords, and parameter operands. The structured use of verbs and keywords in the SQL syntax lets you request data in readable form.

The SQL commands commonly used are:

SELECT
retrieves data from one or more tables. When used in a program, place the query command inside a DECLARE CURSOR command so that you can fetch rows of the query result individually.

INSERT
places a new row in a table.

UPDATE
changes field level data.

DELETE
removes one or more rows from a table.

CREATE TABLE
defines a new table and its columns.

DROP TABLE
erases a table.

ALTER TABLE
adds new columns to a table.

CREATE INDEX
defines an index that lets you access rows of a table in a specific sequence.

DROP INDEX
erases an index.

CREATE VIEW
defines a logical table from one or more tables or views.

DROP VIEW
erases a view definition.

GRANT
grants privileges on a table or view to other users. You can only grant a privilege to other users if you hold the privilege.

DB2 Server for VM operates in two modes:

- **Single User Mode** lets a single application or utility perform in the same virtual machine as DB2 Server for VM. It is used primarily for development and testing. This mode is also intended for dedicated functions like bulk loading and unloading databases, and other situations that may require a dedicated DB2 Server for VM database.

- **Multiple User Mode** lets you and other users or operations access the same database at the same time. It is the most common operational mode. The advantages are shared access and DB2 Server for VM isolation from individual applications through isolation of virtual machines.

Coding SQL Commands

You must place SQL statements in your DB2 Server for VM program that:

- Declare special variables that SQL uses to interact with the host program.
- Declare an SQL Communications Area (SQLCA) to provide for error handling.
- Establish a connection to an DB2 Server for VM database.
- Manipulate the data you need.
- End your logical unit of work (by committing or rolling back the data).
- Release your connection.

Declaring Host Variables to SQL

A ***host variable*** is a variable referenced by SQL in your program. DB2 Server for VM recognizes two types of host variables: ***main variables*** and ***indicator variables***.

Main Variables

Main variables are normal program variables used in SQL statements. To get SQL to recognize these variables, you must place them in a SQL declare section. This is a special area in your program that is delimited by two SQL statements:

- BEGIN DECLARE SECTION
- END DECLARE SECTION.

The length of the main variable names can differ depending on the programming language you are using. The characters of main variable names can consist of A-Z, 0-9, the three national characters (@, #, $), and the underscore. However, you cannot use a number or the underscore as the first character of a variable name. In COBOL, do not give any variable a name beginning with SQL or RDI. In FORTRAN, do not give any variable a name beginning with SQ. These are reserved for DB2 Server for VM use. Other naming restrictions apply to specific languages.

When you reference program variables in SQL statements, preface them with a colon (:). When you reference the same variable in a host language statement, omit the colon. For example, a variable named DBDESC is referenced as :DBDESC in a SELECT command.

Variables used in SQL statements cannot be any of the following:

- Vector or array declarations
- A constant defined by a PARAMETER statement
- Any declarations that use expression to define the length of the variables
- Character variables declared with an undefined length such as CHARACTER*(*).

Indicator Variables

By using optional indicator variables, you can indicate null values on input to DB2 Server for VM (the UPDATE and INSERT statements) or output from DB2 Server for VM (the INTO clause of a FETCH statement). You must declare indicator variables in the SQL declare section. They must be of a host language data type equivalent to the SQL data type SMALLINT. When used in an SQL statement, the indicator variable names must follow the corresponding main variable name and must be preceded with a colon. For example, if the main variable name is DBDESC and the corresponding indicator name is DESCIND, in a SQL statement you would refer to it with the expression :DBDESC:DESCIND.

After an SQL request involving an output variable is satisfied, a value is returned to your program in the indicator variable.

- When the indicator variable value is zero, the value returned into the main variable is not null.
- When the indicator variable value is negative, the main variable is null and should not be used for processing by the host program.
- When the indicator variable value is positive, DB2 Server for VM has truncated the value of the main variable. The returned value was larger than the declared value.

Declaring an SQL Communication Area

Every DB2 Server for VM program must provide a means for handling errors. The SQL Communications Area (SQLCA) provides this. DB2 Server for VM sends messages to SQLCA after executing almost every SQL statement (except declarative statements—BEGIN and END DECLARE SECTION, INCLUDE SQLCA, INCLUDE SQLDA, DECLARE CURSOR, and WHENEVER). Then, using the WHENEVER statement, you can test certain fields of this area for specific conditions during the program's execution. Error handling is important in DB2 Server for VM because it helps protect the integrity of the database when a program fails.

As mentioned previously, when system errors occur, DB2 Server for VM automatically restores all changes made from the start of the logical unit of work up to the point of system failure. When SQL errors occur, your application must tell DB2 Server for VM what action to take. This involves two steps:

- Declaring an SQL Communications Area
- Coding an SQL WHENEVER statement.

To declare an SQL Communications Area, use the following SQL command:

```
INCLUDE SQLCA
```

The WHENEVER statement tests conditions set in the SQLCA by DB2 Server for VM. The WHENEVER statement lets you take specific actions depending on the conditions. The scope of the WHENEVER command is determined by its position in the program rather than its placement in the logical flow. Use the appropriate WHENEVER statements at critical points in your program.

The SQLCA has two especially important fields: the SQLCODE and SQLWARN.

SQLCODE contains a code that indicates the result of each SQL statement. The value in SQLCODE summarizes the execution of your SQL statements:

- When the value is zero, the command has executed successfully.
- When the value is negative, an error condition has occurred (either an error in your program or a system failure).
- When the value is positive, a normal condition (for example, End-Of-File) or a warning condition is indicated.

You can test SQLCODE with the WHENEVER statement. The syntax of this statement is:

```
WHENEVER SQLERROR action
```

Possible actions are `CONTINUE` or `GOTO statement-label`.

When SQLCODE is 100, it indicates a NOT FOUND condition. You can test this condition using the following statement:

```
WHENEVER NOT FOUND action
```

The SQLCA SQLWARNING condition occurs when SQLCODE is greater than 0 but not equal to 100, or the SQL warning indicator, SQLWARN0, contains the value `W`. You can test this condition using the following WHENEVER statement:

```
WHENEVER SQLWARNING action
```

The normal actions are CONTINUE or GOTO statement-label.

For example, in this FORTRAN program, the WHENEVER command causes a branch to statement 90 when an error condition occurs (SQLERROR becomes negative) throughout the program. At statement 90, the WHENEVER is reset to CONTINUE during execution of the ROLLBACK WORK to prevent a failure during ROLLBACK from causing a program loop. After the branch back to statement 10, the WHENEVER branch to 90 is in effect again.

```
      .
      .
      EXEC SQL WHENEVER SQLERROR GO TO 90
      .
      .
      .
10    LASTRC = ISPLNK ('DISPLAY','MENUPAN ')
      .
      .
      .
90    CONTINUE
      EXEC SQL WHENEVER SQLERROR CONTINUE
      EXEC SQL ROLLBACK WORK
      GO TO 10
```

Connecting to DB2 Server for VM

In the z/VM environment, the DB2 Server for VM CONNECT statement is not required to establish a connection between DB2 Server for VM and your program. User ID and password check by z/VM may be sufficient. DB2 Server for VM does implicit connecting for those environments when an explicit connect is not found.

For an explicit connection to DB2 Server for VM, you can use the following statement:

```
EXEC SQL CONNECT :USERID IDENTIFIED BY :PASS
```

:USERID and :PASS are the host variables containing the valid user ID and password needed to execute your application.

Manipulating Data

The body of an DB2 Server for VM application can contain the following DB2 Server for VM commands: SELECT, INSERT, DELETE, UPDATE, CREATE TABLE, DROP TABLE, ALTER TABLE, CREATE INDEX, DROP INDEX, CREATE VIEW, and DROP VIEW. See page “SQL Commands” on page 426 for a description of these commands.

In addition to the DB2 Server for VM commands, the body of your DB2 Server for VM application can also contain host language statements.

Ending Your Logical Unit of Work

The term ***logical unit of work*** means a sequence of SQL commands that DB2 Server for VM views as a unit of consistency and recovery. (These commands can be mixed with non-SQL statements.) This concept is useful because DB2 Server for VM can ensure the integrity of the database. It does this by making sure that either **all** or **none** of the updates in a logical unit of work are done.

A logical unit of work begins with any SQL command and ends with a COMMIT WORK or ROLLBACK WORK command. If a system failure occurs before the explicit end of a logical unit of work, SQL automatically restores all changes made from the start of the logical unit up to the point of system failure. This is called a rollback. However, you must tell DB2 Server for VM what to do for SQL errors.

Releasing the Connection to DB2 Server for VM

To explicitly release the connection to DB2 Server for VM, you can use either of the following statements:

```
EXEC SQL COMMIT WORK RELEASE
```

or

```
EXEC SQL ROLLBACK WORK RELEASE
```

SQL Command Layout

The following is a FORTRAN example showing the structure you need to imbed SQL commands in your application.

```
EXEC SQL BEGIN DECLARE SECTION
   .
   .
   .
  (Variable definitions used by SQL go here.)
   .
   .
   .
EXEC SQL END DECLARE SECTION
EXEC SQL INCLUDE SQLCA
EXEC SQL WHENEVER
EXEC SQL CONNECT :USERID IDENTIFIED BY :PASS
   .
   .
   .
EXEC SQL command-name...
   .
   .
   .
EXEC SQL COMMIT WORK RELEASE
```

Creating DB2 Server for VM Tables

You must have RESOURCE authority to create a table, unless you have a private DBSPACE. If you are not sure that you have RESOURCE authority, speak to your database administrator.

A ***DBSPACE*** is a portion of the database that can contain one or more tables and any associated indexes. Each table stored in DB2 Server for VM is placed in some particular DBSPACE chosen by the creator of the table. The database administrator defines DBSPACEs when the database is generated. Additional spaces can be added later using the ADD DBSPACE function. Each DBSPACE remains as an unnamed "available" DBSPACE until it is "acquired" by means of an ACQUIRE DBSPACE statement. The acquiring user gives a name to the DBSPACE and defines certain characteristics for it (or allows default characteristics).

With DB2 Server for VM, you can define new tables in the database without stopping the system or calling special utilities. You can accomplish this by using the CREATE TABLE SQL data definition statement. The *table-id* operand of the CREATE TABLE statement specifies the table name. As a default, your table name is prefixed with your user ID. The specifications for the table are pairs of column-names and data types with or without the qualifier NOT NULL. This qualifier tells DB2 Server for VM not to allow null values in a particular column. Any statement that later tries to put a null value in that column is rejected with an error code. The optional DBSPACE parameter lets you choose a specific database space in which to create the table.

For example, in a FORTRAN application, the following statement creates a table called NAMELIST in the DBSPACE called TEST.DBSP:

```
      EXEC SQL CREATE TABLE NAMELIST
     1         (FRSTNAME  CHAR(16) NOT NULL,
     2          LASTNAME  CHAR(16) NOT NULL,
     3          SERIALNO  CHAR(6) NOT NULL,
     4          AREACODE  CHAR(3),
     5          ZIPCODE   CHAR(5),
     6          PHNUMBER  CHAR(7))
     7    IN TEST.DBSP
```

Once you create a table, you cannot change the data types of its columns or drop a column from the table. However, you can add new rows to the table using the INSERT command. You can also add new columns to a table using the ALTER TABLE command or drop or delete a table using the DROP TABLE command. You must be the creator of the table or have database administrator authority to delete a table.

Retrieving Data from a Table

To retrieve one row of data from a table, use the SELECT INTO FROM statement. This statement finds one row of the table specified in the FROM clause, selects the columns that were supplied in the select-list, and delivers the results in the host-variables listed in the INTO clause.

To retrieve one or more rows of data from DB2 Server for VM tables, your application should use an SQL ***cursor***. In general terms, a cursor is a pointer to the database. The SQL DECLARE statements define a cursor by associating a name with a query. The query may cause many rows to be returned from the database. These rows are called the ***active set*** (result) of the cursor.

You can manipulate the cursor using the following statements: OPEN, FETCH, PUT, DELETE, UPDATE, and CLOSE.

For example:

```
DECLARE CRSR CURSOR FOR       <---cursor-clause
SELECT FRSTNAME, LASTNAME     <---SELECT-clause
FROM NAMELIST                 <---FROM-clause
WHERE SERIALNO = :EMPSER      <---WHERE-clause
ORDER BY ZIPCODE              <---ORDER-BY-clause
```

To retrieve SQL data, you declare a cursor (CRSR in this example) and associate with it a SELECT statement that describes the information to be retrieved. The SELECT statement must include a SELECT-list that specifies the columns (FRSTNAME, LASTNAME) required and a FROM-list that specifies the table(s) (NAMELIST) that contains those columns. Optionally, a WHERE-clause may filter the results. If it is not provided, all rows qualify for the retrieval. See “Defining Search Conditions” on page 431 for more detail. The optional ORDER BY-clause permits ordering the results of the query. Without it, the ordering is unpredictable.

After you issue the DECLARE CURSOR statement, you must open the cursor with an OPEN statement, using the same cursor-name you specified in the DECLARE CURSOR statement. For example, to open the CRSR cursor, use the following statement in a FORTRAN application:

```
      EXEC SQL OPEN CRSR
```

The OPEN statement determines the active set of the cursor and positions the cursor before the first row of the active set.

Next, to retrieve the data, use the FETCH statement. This tells DB2 Server for VM to advance the cursor to the next row of the active set and to deliver the data into the main variables you specify on the FETCH statement. For example, in a FORTRAN application, the statement:

```
      EXEC SQL FETCH CRSR INTO :FNAME, :LNAME
```

retrieves FRSTNAME and LASTNAME from the NAMELIST table, that matches the conditions set in the query, and puts this information in the variables FNAME and LNAME.

When using the FETCH statement, you must follow certain punctuation rules: Separate the main variables from each other with commas and precede each one with a colon.

When you are finished retrieving data, close the cursor using the CLOSE statement. For example, to close the CRSR cursor, use the following statement in a FORTRAN application:

```
      EXEC SQL CLOSE CRSR
```

Also, the active set becomes undefined when you issue the CLOSE statement.

Defining Search Conditions

To find particular items of data in SQL databases effectively, you need to define ***search conditions*** in the WHERE-clause. These let you control row selection. A search condition is a collection of ***predicates***. A predicate is a comparison of two values or expressions. Along with column names, the expression can

be constants, variables, and any combination of these connected by arithmetic operator. Each predicate specifies a test that DB2 Server for VM applies to the rows of the table.

For example:

```
WHERE SERIALNO = :EMPSER
```

causes DB2 Server for VM to test the values in the SERIALNO column of each row of the NAMELIST table. DB2 Server for VM returns rows to the active set only when the SERIALNO value equals the value in the main variable EMPSER.

`SERIALNO = :EMPSER` is the predicate. `SERIALNO` and `EMPSER` are expressions of the predicate. = is the comparison operator of the predicate.

Comparison Operators

The comparison operators are:

=
Equal to

¬=
Not equal to

>
Greater than

>=
Greater than or equal to

<
Less than

<=
Less than or equal to

For example:

```
PARTNO > 105
```

If you use variables in an expression, you must precede the variable name with a colon. This distinguishes it from a column name. Thus, the predicate:

```
SERIALNO > :EMPSER
```

means the value in column SERIALNO is greater than the value in variable EMPSER.

Conversely, the predicate:

```
:EMPSER  >  SERIALNO
```

means the value in variable EMPSER is greater than the value in column SERIALNO.

You can use constants within expressions, using any data types the language supports, but with some exceptions.

Arithmetic Operators

The four arithmetic operators are:

+
Addition

-
Subtraction

*****
Multiplication

/
Division.

You can use parentheses in an expression if you want to establish precedence among the operators. The default order of precedence is from right to left:

1. negations
2. multiplication or division
3. addition or subtraction.

Logical Operators

The logical operators are: NOT, AND, and OR.

You can use the logical operator NOT to negate a predicate. For example:

```
NOT ZIPCODE = 90023
```

You can connect predicates with the logical operators AND and OR:

```
AREACODE = 213 AND ZIPCODE = 90021 OR ZIPCODE = 90022
```

The order of precedence for the logical operators is:

1. NOT
2. AND
3. OR.

In the preceding example, the statement is true when AREACODE = 213 and ZIPCODE = 90021 **or** when ZIPCODE = 90022, regardless of the value of AREACODE.

By using parentheses, you can override this order. If you want to select data only when AREACODE equals 213 and ZIPCODE equals **either** 90021 **or** 90022, you can code:

```
AREACODE = 213 AND (ZIPCODE = 90021 OR ZIPCODE = 90022)
```

Because the AND is evaluated before the OR, this is equivalent to:

```
AREACODE = 213 AND ZIPCODE = 90021 OR ZIPCODE = 90022 AND AREACODE = 213
```

Defining Additional Predicates

SQL provides four additional types of predicates you can use in search conditions. You can use them in addition to the standard ones that compare two expressions. These predicates are:

BETWEEN
determines if the value of an expression lies between the values of two other expressions. For example:

```
ZIPCODE BETWEEN :LIM1 AND :LIM2
```

This is equivalent to:

```
:LIM1 <= ZIPCODE <= :LIM2
```

IN
compares the value of an expression with a list of items. The predicate is satisfied if the expression equals any item listed. For example:

```
ZIPCODE IN (90021, :P2, :P3, :P4)
```

IS NULL
explicitly looks for null values in tables (empty fields) or exclude null values from consideration. For example:

```
ZIPCODE IS NULL
```

LIKE
searches for character string data that partially matches a given string. For example:

```
FRSTNAME LIKE "%ANNE%"
```

This example is met by values such as "ROXANNE", "ANNETTE", and "JANNER" as well as by "ANNE". The percent sign (%) represents a wild-card character and means any string of zero or more characters.

You can prefix any of these predicates with the logical operator NOT.

Using Built-In SQL Functions

SQL has two types of built-in functions: column functions and scalar functions. Column functions apply the function to a group of values in a column and produce one result. Scalar functions apply the function to one or more values in each row and produce a result for each row.

The following is an example of an SQL column function, AVG, used in a FORTRAN application. To obtain the average of the values found in the QUANTITY column from the NAMELIST table, use the following statements:

```
      EXEC SQL DECLARE CRSR CURSOR FOR
     1    SELECT AVG(QUANTITY)
     2    FROM NAMELIST
```

Excluding Duplicates

The keyword ALL causes every value that satisfies the expression to be selected. This is the default. The keyword DISTINCT limits the selection to a single match.

For example, to get a list of different surnames, in a FORTRAN application, you would use an expression such as:

```
      EXEC SQL DECLARE CRSR CURSOR FOR
     1    SELECT DISTINCT LASTNAME
     2    FROM NAMELIST
```

Manipulating Data in a DB2 Server for VM Table

There are SQL data manipulation statements that let you insert new rows into tables or delete or update existing rows. Here are the three data manipulation statements:

- **INSERT** lets you insert one new row into a given table. Also, by using the SELECT clause, you can insert several new rows selected or computed from other tables. You can insert data into any table you create. You can also insert data into another user's table, if you have INSERT privilege on that table.

  For example, in a FORTRAN application, the statements:

  ```
       EXEC SQL INSERT INTO NLIST
     1  SELECT LNAME,FNAME,SERIALNO
     2  FROM NAMELIST
     3  WHERE SERIALNO = :EMPSER
  ```

  inserts into table NLIST columns LNAME, FNAME, and SERIALNO of all the rows of table NAMELIST having the SERIALNO column equal to the value in the host variable EMPSER.

- **DELETE** deletes one or more rows from a given table. However, first you must specify a selection criterion (WHERE clause). See "Defining Search Conditions" on page 431 for details on defining search conditions. Otherwise, the DELETE statement deletes all table rows and sets a warning indicator (SQLWARN4). You can test the value of SQLWARN4 and, in case of error, issue the ROLLBACK WORK command. You can delete rows from any table you create. You can also delete data from another user's table, if you have DELETE privilege on that table.

  For example, in a FORTRAN application, the statement:

  ```
        EXEC SQL DELETE FROM NAMELIST WHERE SERIALNO = :EMPSER
  ```

  deletes the row or rows having the SERIALNO column equal to the value in the host variable EMPSER.

  You can also delete the row that the current cursor points to by specifying WHERE CURRENT OF *cursor-name*.
- **UPDATE** changes the value of one or more fields in a table. You can update rows in any table you create. You can also update data in another user's tables, if you have the UPDATE privilege on the columns of that table.

  You can also change the value of one or more fields in a table by specifying WHERE CURRENT OF *cursor-name*.

Creating Views in DB2 Server for VM

DB2 Server for VM can create **views** of a table. This is one of its most useful facilities. Views let you and other users see different presentations of the same data.

For example, if your NAMELIST table contains employee salaries, you may want to restrict access to that data. Other users may need to see salaries but not addresses, and so on. Each user can have a different view of the data in the NAMELIST table. Each view appears to be a table and has its own name.

Views are based on tables, but views are not stored as physical tables. However, there are some restrictions on views that real tables do not have.

The CREATE VIEW statement creates views. (You must have SELECT privilege for the underlying table.)

For example, in a FORTRAN application, the following statement:

```
      EXEC SQL CREATE VIEW AREA213 (FNAME, LNAME, EMPSER, PHONE) AS
     1    SELECT FRSTNAME, LASTNAME, SERIALNO, PHNUMBER
     2    FROM NAMELIST
     3    WHERE AREACODE = 213
```

creates a view called AREA213 containing the names, serial number, and phone number of the employees living in area code 213. Its four columns have names distinct from the corresponding names in the NAMELIST table. If these names are not specified, DB2 Server for VM takes them from the original table.

Your application can now insert data into view AREA213, update data to view AREA213, and delete data from view AREA213. These changes will be applied to the actual NAMELIST table.

Remember, there are certain restrictions on views. See the appropriate DB2 Server for VM publication for details.

When you finish with a view, you can drop it using the DROP VIEW statement. For example, to drop view AREA213, use the following statement in a FORTRAN application:

```
      EXEC SQL DROP VIEW AREA213
```

Preprocessing Your DB2 Server for VM Application

After you write your SQL application, but before you compile and run your application, SQL must ***preprocess*** the application.

Preprocessing does two things: First, it modifies the source program by converting SQL statements into valid programming language statements. (SQL statements are kept as comments.) There is a separate DB2 Server for VM preprocessor for each language. The source program is then ready for normal processing. Second, it optimizes and compiles the SQL statements by defining them to DB2 Server for VM and creating an access module that efficiently executes the SQL requests that the program makes. The access module is created and stored in a DB2 Server for VM database.

During DB2 Server for VM preprocessing, DB2 Server for VM analyzes and converts the embedded SQL commands to DB2 Server for VM calls before compilation. That is, DB2 Server for VM chooses the best access path for each SQL command in the program, based on indexes and data statistics, for example, and stores the access information in the access module. When DB2 Server for VM loads the access module, it checks that the module is still valid. An access module may be invalid, for example, if a path is based on an index that is no longer available.

When you run a program, the access module created by the DB2 Server for VM preprocessor is called to handle each SQL command.

DB2 Server for VM provides some special programs that your application needs to link to at time of execution. These DB2 Server for VM provided programs, along with the DB2 Server for VM EXECs that are needed to identify the DB2 Server for VM database and start the DB2 Server for VM preprocessors, are stored on the DB2 Server for VM production minidisk. You must access this disk in order to use DB2 Server for VM.

Figure 73 on page 436, followed by a step-by-step procedure, describes how to create an executable SQL application written in COBOL.

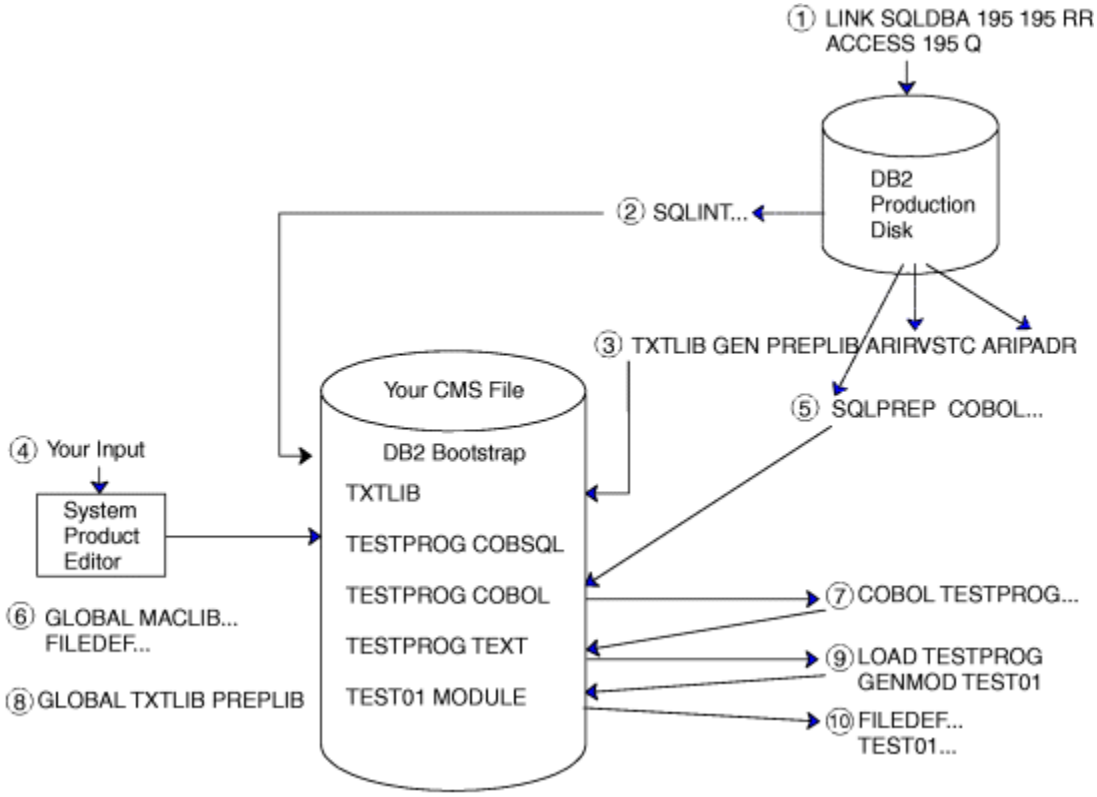


Figure 73. Creating an Executable SQL Program

1. Access the DB2 Server for VM production minidisk:

   ```
   LINK SQLDBA 195 195 RR
   ACCESS 195 Q
   ```

   The production minidisk is established during the DB2 Server for VM installation process. It contains the DB2 Server for VM EXECs and programs required at execution time. These programs must be linked with your program in a later step.

2. Identify the DB2 Server for VM database. To do this, you will use the DB2 Server for VM EXEC, SQLINIT. It names the database and stores bootstrap information for that database on your A-disk. Because this information is on your A-disk, you need only do this step once (even if you log off), unless you subsequently need to change to another DB2 Server for VM database. An example of this EXEC is:

   ```
   SQLINIT DBNAME(DBASE01)
   ```

   DBASE01, in this example, is the name of the DB2 Server for VM database selected.

3. (Step 8 requires that you have established a CMS TXTLIB that contains the execution time DB2 Server for VM programs for linking with your program.) Use the TXTLIB command to create the TXTLIB that contains the DB2 Server for VM programs ARIRVSTC and ARIPADR. ARIPADR and ARIRVSTC are required for COBOL. This step need only be done once (even if you log off), because the TXTLIB is stored on your A-disk. Enter:

   ```
   TXTLIB
   GEN PREPLIB ARIRVSTC ARIPADR
   ```

4. Use XEDIT to build your program. The file type must be COBSQL if it is written in COBOL. These are the file types required for the SQL preprocessors in the next step.

5. The DB2 Server for VM preprocessors are invoked by the DB2 Server for VM EXEC, SQLPREP. The following are examples of invoking SQLPREP:

   ```
   SQLPREP COBOL PREPPARM(PREPNAME=TESTPROG,QUOTE)
           SYSIN(TESTPROG) SYSPUNCH(TESTPROG) SYSPRINT(PRINTER)
   ```

   - The first parameter identifies the programming language. This selects the particular preprocessor and is followed by the parameters to that preprocessor.
   - PREPPARM has several subparameters. The main subparameter is PREPNAME. This is generally the same as the name that you have assigned to your program. For COBOL, you may want to use the keyword subparameter, QUOTE, to indicate that you are going to use the QUOTE option for the COBOL compiler. The QUOTE parameter (or APOST, the default) has no affect on the coding of SQL statements in the COBOL program, but informs the SQL preprocessor what to expect as delimiters for COBOL strings.
   - The SYSIN parameter specifies the file name of the input source program.
   - The SYSPUNCH parameter specifies the file name of the output of the preprocess step, which is normally the same as specified for SYSIN. The default file type assigned by DB2 Server for VM is the file type required for the associated compiler.
   - The SYSPRINT parameter specifies the file name (default file type is LISTPREP) for receiving the printed output of the preprocessor. In this example, it is directed to the virtual printer, rather than a CMS file.

6. The CMS GLOBAL command and FILEDEF command identify the MACLIBs and workfiles required by the particular compiler. Enter:

   ```
   GLOBAL MACLIB COBOLVS DMSGPI FILEDEF ...  (work files)
   ```

7. Start the appropriate compiler. The input files to the compiler must have the appropriate file type (COBOL, in this example). The compiler produces a TEXT file. Enter:

   ```
   COBOL2 TESTPROG
   ```

8. The CMS GLOBAL command identifies the text library needed to execute the application. Step 3 created the text library, PREPLIB, that is used here.

   ```
   GLOBAL TXTLIB PREPLIB
   ```

9. The CMS LOAD command loads the new application text file into storage and to link it with the required DB2 Server for VM programs from PREPLIB. The CMS GENMOD command generates a module on the A-disk for the application program and assigns it the name TEST01.

```
LOAD TESTPROG
GENMOD TEST01
```

10. After issuing all the FILEDEFs that may be required for the application program, invoke the new module for execution.

```
TEST01
```

Using SQL Interactively

DB2 Server for VM also includes the Interactive Structured Query Language (ISQL) facility that lets you enter SQL commands directly from your terminal. It is also useful for prototyping commands that you plan to use in your programs. ISQL also simplifies data handling by offering facilities for querying data, manipulating data, and writing reports.

For more information on ISQL, see the *DB2 Server for VM ISQL Guide and Reference for IBM VM Systems*.

You can also use the Query Management Facility (QMF) to enter SQL queries from your terminal. In addition to entering and processing your SQL queries, QMF allows you to format your report, display your report as a chart, timestamp your report, and manipulate data.

For details on how to use QMF, see *Using QMF*.

Chapter 29. Using Data Compression Services

This chapter covers the following topics:

- Compression Processing
- Expansion Processing
- Using Compression and Expansion Dictionaries
- Compressing and Expanding CMS Data.

Compression and Expansion Services

You can save data in a compressed format to conserve storage media and network transmission line costs. The CSRCMPSC macro provides a pair of services that compress and expand data. These services are available when the CVTCMPSC bit is on in the communication vector table (CMSCVT).

Compression takes an input string of data and, using a data area called a *dictionary*, produces an output string of compression symbols. Each symbol represents a string of one of more characters from the input.

Expansion takes an input string of compression symbols and, using a *dictionary*, produces an output string of the characters represented by those compression symbols.

Parameters for the CSRCMPSC macro are in an area mapped by DSECT CMPSC of the CSRYCMPS macro and specified by the CBLOCK parameter of the CSRCMPSC macro. This area contains the following information:

- The address, ALET, and length of a source area. The source area contains the data to be compressed for a compression operation, or to be expanded for an expansion operation.
- The address, ALET, and length of a target area. After the macro runs, the target area contains the compressed data for a compression operation, or the expanded data for an expansion operation.
- An indication of whether to perform compression or expansion.
- The address and format of a dictionary to be used to perform the compression or expansion. The dictionary must be in the same address space as the source area.

Compressing and expanding data is described in the following topics:

- “Compression and Expansion Dictionaries” on page 439
- “Compression Processing” on page 440
- “Expansion Processing” on page 441
- “Dictionary Entries” on page 441
- “Building the CSRYCMPS Area” on page 450
- “Determining if the CSRCMPSC Macro Can Be Issued on a System” on page 453

Compression and Expansion Dictionaries

To use the Data Compression Services for compression and expansion, the CSRCMPSC macro uses two dictionaries: the compression dictionary and the expansion dictionary. These dictionaries are logically and physically related. When you expand the data that has been compressed, you want the result to match the original data. Thus the dictionaries are complementary. When compression is being done, the expansion dictionary must immediately follow the compression dictionary, because the compression algorithm examines entries in the expansion dictionary.

Each dictionary consists of 512, 1024, 2048, 4096, or 8192 8-byte entries and begins on a page boundary. When the system determines or uses a compression symbol, the symbol is 9, 10, 11, 12, or 13 bits long, with the length corresponding to the number of entries in the dictionary. You will have

to specify the size of the dictionary in the CMPSC_SYMSIZE field of the CSRYCMPS mapping macro as follows:

SYMSIZE
Meaning

1
Symbol size 9 bits, dictionary has 512 entries

2
Symbol size 10 bits, dictionary has 1024 entries

3
Symbol size 11 bits, dictionary has 2048 entries

4
Symbol size 12 bits, dictionary has 4096 entries

5
Symbol size 12 bits, dictionary has 8192 entries

Using Compression and Expansion Services

To help you use the compression services, the S-disk contains the following compiled REXX execs:

- CSRBDICV for building compression and expansion dictionaries
- CSRCMPEV to run a test, compress, and re-expand files using the abstract dictionary created by the CSRBDICV EXEC. Reports are generated giving statistics on the efficiency of the compress and expand functions with the current dictionary set.

For information on how to use these execs, see the *z/VM: CMS Commands and Utilities Reference*. For additional information about compression and using the execs, see *Enterprise Systems Architecture/390 Data Compression*.

Compression Processing

The compression dictionary consists of a specified number of 8-byte entries. The first 256 dictionary entries correspond to the 256 possible values of a byte and are referred to as *alphabet entries*. The remaining entries are arranged in a downward tree, with the alphabet entries being the topmost entries in the tree. That is, an alphabet entry may be a *parent entry* and contain the index of the first of one or more contiguous *child entries*. A child entry may, in turn, be a parent and point to its own children. Each entry may be identified by its *index*, meaning the positional number of the entry in the dictionary; the first entry has an index of 0.

An alphabet entry represents one character. A nonalphabet entry represents all of the characters represented by its ancestors and also one or more additional characters called *extension characters*. For compression, the system uses the first character of an input string as an index to locate the corresponding alphabet entry. Then the system compares the next character or characters of the string against the extension character or characters represented by each child of the alphabet entry until a match is found. The system repeats this process using the children of the last matched entry, until the last possible match is found, which might be a match on only the alphabet entry. The system uses the index of the last matched entry as the compression symbol.

The first extension character represented by a child entry exists as either a child character in the parent or as a sibling character. A parent can contain up to four or five child characters. If the parent has more children than the number of child characters that can be in the parent, a dictionary entry named a *sibling descriptor* follows the entry for the last child character in the parent. The sibling descriptor can contain up to six additional child characters, and a dictionary entry named a sibling descriptor extension can contain eight more child characters for a total of fourteen. These characters are called *sibling characters*. The corresponding additional child entries follow the sibling descriptor. If necessary, another sibling descriptor follows the additional child entries, and so forth. The dictionary entries that are not sibling descriptors or sibling descriptor extensions are called character entries.

If a nonalphabet character entry represents more than one extension character, the extension characters after the first are in the entry; they are called additional extension characters. The first extension character exists as a child character in the parent or as a sibling character in a sibling descriptor or sibling descriptor extension. The nonalphabet character entries represent either:

- If the entry has no children or one child, from one to five extension characters.
- If the entry has more than one child, one or two extension characters. If the entry represents one extension character, it can contain five child characters. If it represents two extension characters, it can contain four child characters.

Expansion Processing

The dictionary used for expansion also consists of a specified number of 8-byte entries. The two types of entries used for expansion are:

- Unpreceded entries
- Preceded entries

The compression symbol, which is an index into the dictionary, locates that index's dictionary entry. The symbol represents a character string of up to 260 characters. If the entry is an unpreceded entry, the expansion process places at offset 0 from the current processing point the characters designated by that entry. Note that the first 256 correspond to the 256 possible values of a byte and are assumed to designate only the single character with that byte value.

If the entry is a preceded entry, the expansion process places the designated characters at the specified offset from the current processing point. It then uses the information in that entry to locate the preceding entry, which may be either an unpreceded or a preceded entry, and continues as described previously.

The sibling descriptor extension entries described earlier are also physically located within the expansion dictionary.

Dictionary Entries

The following notation is used in the diagrams of dictionary entries:

{cc}
Character may be present

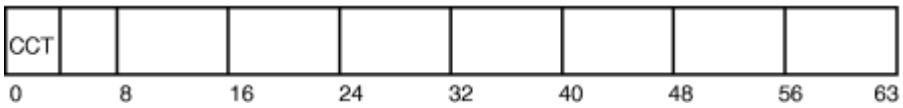
...
The preceding field may be repeated

Compression Dictionary Entries

Compression entries are mapped by DSECTs in macro CSRYCMPD.

The first four entries that follow give the possible values for bits 0-2, which are designated CCT.

Character Entry Generic Form (DSECT CMPSCDICT_CE)



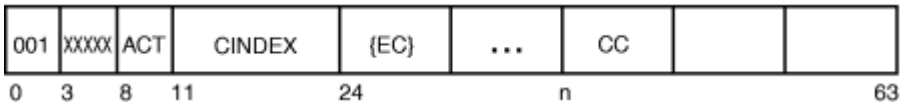
CCT
A 3-bit field (CMPSCDICT_CE_CHILDCT) specifying the number of children. The total number of children plus additional extension characters is limited to 5. If this field plus the number of additional characters is 6, it indicates that, in addition to the maximum number of children for this entry, there is a sibling descriptor entry that describes additional children. The sibling descriptor entry is located at dictionary entry CMPSCDICT_CE_FIRSTCHILDINDEX plus the value of CMPSCDICT_CE_CHILDCT. The value of CMPSCDICT_CE_CHILDCT plus the number of additional extension characters must not exceed 6.

Character Entry CCT=0 (DSECT CMPSCDICT_CE)

ACT
A 3-bit field (CMPSCDICT_CE_AECCT) indicating the number of additional extension characters in the entry. Its value must not exceed 4. This field must be 0 in an alphabet entry.

EC
An additional extension character. The 5-character field CMPSCDICT_CE_CHILDCHAR is provided to hold the additional extension characters followed by the child characters.

Character Entry CCT=1 (DSECT CMPSCDICT_CE)



XXXXX
A 5-bit field (CMPSCDICT_CE_EXCHILD) with the first bit indicating whether it is necessary to examine the character entry for the child character (looking either for additional extension characters or more children). The other bits are ignored when CCT=1.

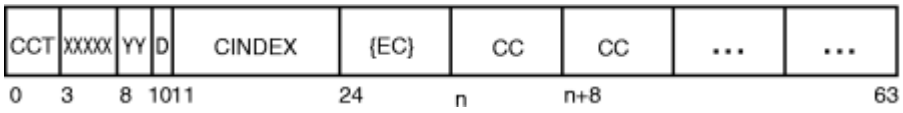
ACT
A 3-bit field (CMPSCDICT_CE_AECCT) indicating the number of additional extension characters. Its value must not exceed 4. This field must be 0 in an alphabet entry.

CINDEX
A 13-bit field (CMPSCDICT_CE_FIRSTCHILDINDEX) indicating the index of the first child. The index for child *n* is then CMPSCDICT_CE_FIRSTCHILDINDEX + *n*-1.

EC
An additional extension character. The 5-character field CMPSCDICT_CE_CHILDCHAR is provided to hold the additional extension characters followed by the child characters.

CC
Child character, at bit *n* = 24 + (ACT * 8). The 5-character field CMPSCDICT_CE_CHILDCHAR is provided to hold the additional extension characters followed by the child characters.

Character Entry CCT>1 (DSECT CMPSCDICT_CE)



CCT
A 3-bit field (CMPSCDICT_CE_CHILDCT) specifying the number of children. For this case, because CCT>1, the range for CCT is 2 to 6 if D=0 or 2 to 5 if D=1. If this field plus the value of D is 6, it indicates that, in addition to the maximum number of children for this entry (4 if D=1, 5 if D=0), there is a sibling descriptor entry that describes additional children. The sibling descriptor entry is located at dictionary entry CMPSCDICT_CE_FIRSTCHILDINDEX plus the value of CMPSCDICT_CE_CHILDCT.

XXXXX
A 5-bit field (CMPSCDICT_CE_EXCHILD) with a bit for each child in the entry. The field indicates whether it is necessary to examine the character entry for the child character (looking either for additional extension characters or more children). The bit is ignored if the child does not exist.

YY
A 2-bit field (CMPSCDICT_CE_EXSIB) providing examine-child bits for the 13th and 14th siblings designated by the first sibling descriptor for children of this entry. The bit is ignored if the child does

not exist. Note that this is a subfield of CMPSCDICT_CE_AECCT. Do not set both this field and field CMPSCDICT_CE_AECCT in a character entry.

D
A 1-bit field (CMPSCDICT_CE_ADDEXTCHAR) indicating whether there is an additional extension character. Note that this is a subfield of CMPSCDICT_CE_AECCT. Do not set both this field and field CMPSCDICT_CE_AECCT in a character entry. This bit must be 0 in an alphabet entry.

CINDEX
A 13-bit field (CMPSCDICT_CE_FIRSTCHILDINDEX) indicating the index of the first child. The index for child *n* is CMPSCDICT_CE_FIRSTCHILDINDEX + *n*-1.

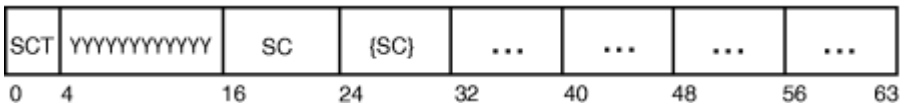
EC
An additional extension character. The 5-character field CMPSCDICT_CE_CHILDCHAR is provided to hold the additional extension character followed by the child characters. There is no additional extension character if D=0.

CC
Child character. The 5-character field CMPSCDICT_CE_CHILDCHAR is provided to hold the additional extension characters followed by the child characters. The first child character is at bit *n* = 24 + (D * 8).

Alphabet Entries (DSECT CMPSCDICT_CE)

The alphabet entries have the same mappings as character entries but without the additional extension characters. The character entries are "Character Entry Generic Form (DSECT CMPSCDICT_CE)" on page 441, "Character Entry CCT=0 (DSECT CMPSCDICT_CE)" on page 442, "Character Entry CCT=1 (DSECT CMPSCDICT_CE)" on page 442, and "Character Entry CCT>1 (DSECT CMPSCDICT_CE)" on page 442.

Format 1 Sibling Descriptor (DSECT CMPSCDICT_SD)



SCT
A 4-bit field (CMPSCDICT_SD_SIBCT) specifying the number of sibling characters. The number of sibling characters is limited to 14. If this field is 15, it indicates that there are 14 sibling characters associated with this entry and that there is another sibling descriptor entry, which describes additional children. That sibling descriptor entry is located at dictionary entry this-sibling-descriptor-index + 15. If there are 1 to 6 sibling characters, they are contained in this entry, and the dictionary entries for those characters are located at this-sibling-descriptor-index + *n*, where *n* is 1 to 6. If there are 7 to 14 sibling characters, the first 6 are as described above, and characters 7 through 14 are located in the **expansion** dictionary entry. (See "Sibling Descriptor Extension Entry (DSECT CMPSCDICT_SDE)" on page 444.) The index of the character entry is this-sibling-descriptor-index. The number of sibling characters should not be 0.

YYYYYYYYYYYY
A 12-bit field (CMPSCDICT_SD_EXSIB), one for each sibling character, indicating whether to examine the character entries for sibling characters 1 through 12. Recall that the examine-sibling indicator for sibling characters 13 and 14 for the first sibling descriptor is in the character entry field CMPSCDICT_CE_EXSIB. If this is not the first sibling descriptor for the child entry, then the character entries for sibling characters 13 and 14 are examined irregardless. The bit is ignored if the sibling does not exist.

SC
Sibling character. Sibling characters 8 through 14 are in the expansion dictionary. (See "Sibling Descriptor Extension Entry (DSECT CMPSCDICT_SDE)" on page 444.) The 6-character field (CMPSCDICT_SD_CHILDCHAR) is provided to contain the sibling characters. The index of the character entry for sibling character *n* is this-sibling-descriptor-index + *n*-1.

Note: Data Compression Services (CSRCMPSC) only supports the use of format 1 sibling descriptors. However, when using the CMPSC hardware instruction directly (for example, not through the macro

call), dictionaries may contain format 0 sibling descriptors. For more information on format 0 sibling descriptors, see *Enterprise Systems Architecture/390 Data Compression*.

Expansion Dictionary Entries

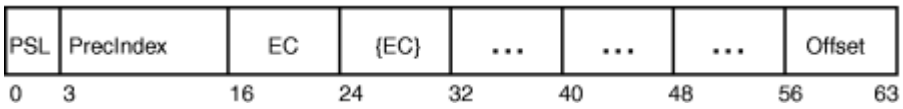
Expansion entries are mapped by DSECTs in macro CSRYCMPD.

Unpreceded Entry (DSECT CMPSCDICT_UE)

CSL
A 3-bit field (CMPSCDICT_UE_COMPSYMLEN) indicating the number of characters contained in CMPSCDICT_UE_CHARS. These characters will be placed at offset 0 in the expanded output. This field should not have a value of 0.

EC
Expansion character. The 7-character field (CMPSCDICT_UE_CHARS) is provided to contain the expansion characters.

Preceded Entry (DSECT CMPSCDICT_PE)



PSL
A 3-bit field (CMPSCDICT_PE_PARTSYMLEN) indicating the number of characters contained in CMPSCDICT_PE_CHARS. These characters will be placed at the offset indicated by CMPSCDICT_PE_OFFSET in the expanded output. This field must not be 0, because 0 indicates an unpreceded entry.

PrecIndex
A 13-bit field (CMPSCDICT_UE_PRECENTINDEX) indicating the index of the dictionary entry with which processing is to continue.

EC
Expansion character. The 5-character field (CMPSCDICT_PE_CHARS) is provided to contain the expansion characters.

Offset
A 1-byte field (CMPSCDICT_PE_OFFSET) indicating the offset in the expanded output for characters in CMPSCDICT_PE_CHARS.

Sibling Descriptor Extension Entry (DSECT CMPSCDICT_SDE)

SC
Sibling character. The 8-character field (CMPSCDICT_SDE_CHARS) is provided to contain the sibling characters. The *n*th sibling character in this entry is actually overall sibling character number 6 + *n*, because the first 6 characters were contained in the corresponding sibling descriptor entry. The index of the character entry for the *n*th character is this-sibling-descriptor-index + 6 + *n*-1.

Dictionary Restrictions

Set up the compression dictionary so that:

- The algorithm does not create a compression symbol that represents a string of more than 260 characters.
- No character entry has more than 260 total children, including all sibling descriptors for that character entry.
- No character entry has a child count greater than 6.
- No character entry has more than 4 additional extension characters when there are 0 or 1 child characters.
- No sibling descriptor indicates 0 sibling characters.

Set up the expansion dictionary so that:

- Expansion of a compression symbol does not use more than 127 dictionary entries.

Other Considerations

If the first child character matches, but its additional extension characters do not match and the next child character is the same as the first, the system continues compression match processing to try to find a compression symbol that contains that child character. If, however, the next child character is not the same, compression processing uses the current compression symbol as the result. You can set up the child characters for an entry to take advantage of this processing.

If a parent entry does not have the examine child bit (CMPSCDICT_CE_EXCHILD) on for a particular child character, then the child character entry should not have any additional extension characters or children. The system will not check the entry itself for additional extension characters or children.

If a parent or sibling descriptor entry does not have the examine sibling bit (CMPSCDICT_CE_EXSIB) on for a particular sibling character, then the character entry for that sibling character should not to have any additional extension characters or children. The system will not check the entry itself for additional extension characters or children.

Compression Dictionary Examples

In the following examples, most fields contain their hexadecimal values. However, for clarity, the examine-child bit fields are displayed with their bit values.

Example 1

The dictionary looks as follows:

Hex Entry
Description

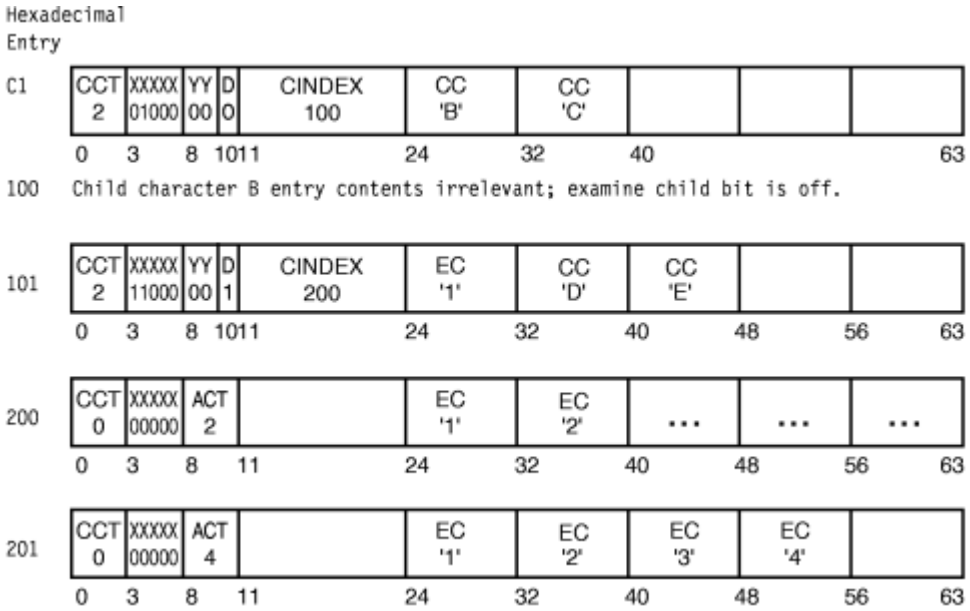
C1
Alphabet entry for character A; 2 child characters B and C. The first child index is X'100'.

100
Entry for character B; no additional extension characters; no children.

101
Entry for character C; additional extension character 1; 2 child characters D and E. The first child index is X'200'.

200
Entry for character D; 2 additional extension characters 1 and 2; no children.

201
Entry for character E; 4 additional extension characters 1, 2, 3, and 4; no children.



If the input string is AD, the output string will consist of 2 compression symbols: one for A and one for D. When examining the dictionary entry for character A, the system determines that none of A's children match the next input character, D, and so returns the compression symbol for A. When examining the dictionary entry for character D, the system determines that it has no children, and so returns the compression symbol for D.

If the input string is AB, the output string will consist of 1 compression symbol for both input characters. When examining the dictionary input for character A, the system determines that A's first child character matches the next input character, B, and so looks at entry X'100'. Because that entry has no additional extension characters, a match is determined. Because there are no further input characters, the scan concludes.

If the input string is AC, the output string will consist of 2 compression symbols: one for A and one for C. When examining the dictionary input for character A, the system determines that A's second child character matches the next input character, C, and so looks at entry X'101'. Because that entry has an additional extension character, but the input string does not contain this character, no match is made, and the output is the compression symbol for A. Processing character C results in the compression symbol for C.

If the input string is AC1, the output string will consist of 1 compression symbol. When examining the dictionary input for character A, the system determines that A's second child character matches the next input character, C, and so looks at entry X'101'. Because that entry has an additional extension character, and the input string does contain this character, 1, a match is made, and the output is the compression symbol for AC1.

Similarly, the set of input strings longer than one character compressed by this dictionary are:

Hex Symbol
String

100
AB

101
AC1

200
AC1D12

201
AC1E1234

The compression symbol is the index of the dictionary entry. Based on this, you can see that the expansion dictionary must result in the reverse processing; for example, if a compression symbol of

X'201' is found, the output must be the string AC1E1234. See “Expansion Dictionary Example” on page 449 for expansion dictionary processing.

Example 2

In this example, there are more than 5 children and the dictionary looks like the following:

Hex Entry
Description

C2
Alphabet entry for character B; child count of 6 (indicating 5 children plus a sibling descriptor); first child index is X'400', children are 1, 2, 3, 4, and 5.

400
Entry for character 1; no additional extension characters; no children.

401-404
Entries for characters 2 through 5; no additional extension characters; no children.

405
Sibling descriptor; child count of 15, which indicates 14 children plus another sibling descriptor; sibling characters A, B, C, D, E, and F.

405
Sibling descriptor extension. In the **expansion dictionary entry X'405'**, the sibling characters are G, H, I, J, K, L, M, and N.

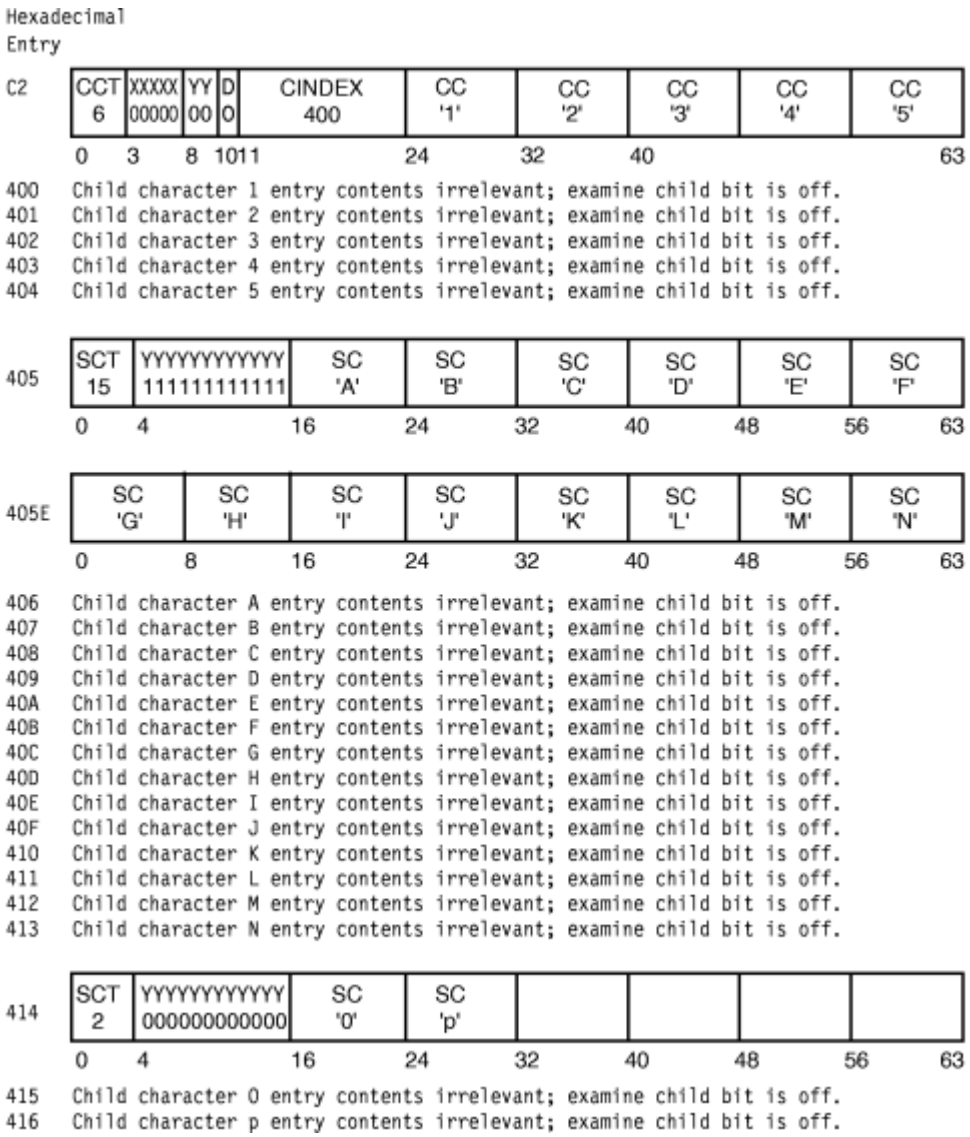
406
Entry for character A; no additional extension characters; no children.

407-413
Entries for characters B through N; no additional extension characters; no children.

414
Next sibling descriptor; child count of 2; child characters O and P.

415
Entry for character O; no additional extension characters; no children.

416
Entry for character P; no additional extension characters; no children.



The set of input strings longer than one character compressed by this dictionary are:

Hex Symbol
String

400-404
B1, B2, B3, B4, B5

406-40B
BA, BB, BC, BD, BE, BF

40C-413
BG, BH, BI, BJ, BK, BL, BM, BN

415-416
BO, BP

There are no compression symbols for 405 and 414. These are the sibling descriptor entries. Because their sibling descriptor extensions are located at those indices in the expansion dictionary (not the preceded or unpreceded entries required for expansion), it is important that no compression symbol have that value.

Example 3

In this example, the children have the same value and the dictionary looks like the following:

Hex Entry
Description

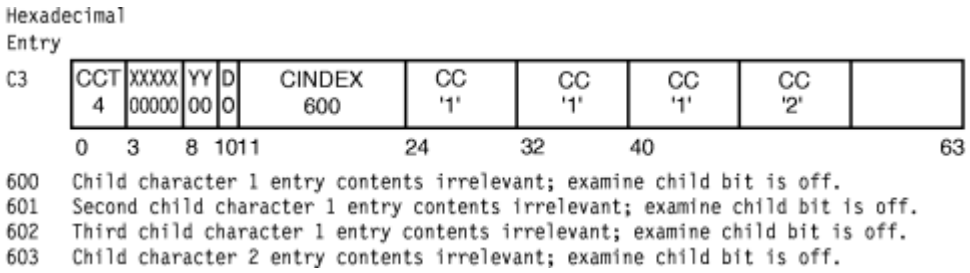
C3
Alphabet entry for character C; child count of 4. The first child index is X'600' and the child characters are 1, 1, 1, and 2.

600
Entry for character 1; 4 additional extension characters A, B, C, and D; no children.

601
Entry for character 1; 3 additional extension characters A, B, and C; no children.

602
Entry for character 1; 2 additional extension characters A and B; no children.

603
Entry for character 2; no additional extension characters; no children.



The set of input strings longer than one character compressed by this dictionary are:

Hex Symbol
String

600
C1ABCD

601
C1ABC

602
C1AB

603
C2

By taking advantage of the special processing when the second and subsequent child characters match the first, you can reduce the number of dictionary entries searched to determine the compression symbols. For example, to find that X'601' is the compression symbol for the characters C1ABC, the processing examines entry X'C3', then entry X'600' then entry X'601' Entry X'600' does not match because the input string does not have all 4 extension characters. There are alternate ways of setting up the dictionary to compress the same set of input strings handled by this dictionary.

Expansion Dictionary Example

Example

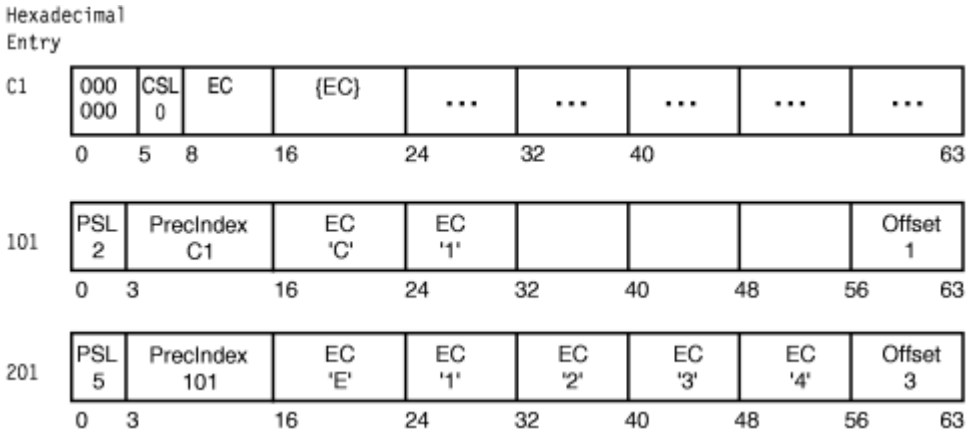
Suppose the expansion dictionary looks like the following:

Hex Entry
Description

C1
Alphabet entry for character A. This by definition is an unpreceded entry.

101
A preceded entry, with characters C and 1; with preceding entry index of X'C1'; offset of 1.

201
A preceded entry, with characters E, 1, 2, 3, and 4; with preceding entry index of X'101'; offset of 3.



When processing an input compression symbol of X'201':

- Characters E1234 are placed at offset 3, and processing continues with entry X'101'.
- Characters C1 are placed at offset 1, and processing continues with entry X'C1'.
- Character A is placed at offset 0.

The expansion results in the 8 characters A, C, 1, E, 1, 2, 3, and 4 placed in the output string.

Building the CSRYCMPS Area

The CSRYCMPS area is mapped by the CSRYCMPS mapping macro and is specified in the CBLOCK parameter of the CSRCMPSC macro. The area consists of 9 words that should begin on a word boundary. Unused bits in the first word must be set to 0.

- Set 4-bit field CMPSC_SYMSIZE in byte CMPSC_FLAGS_BYTE2 to a number from 1 to 5 to indicate both the number of entries in the dictionary and the size of a compressed symbol.
- If expanding, turn on bit CMPSC_EXPAND in byte CMPSC_FLAGS_BYTE2. Otherwise, make sure that the bit is off.
- Set field CMPSC_DICTADDR to the address of the necessary dictionary. If compressing, this should be the compression dictionary, which must be immediately followed by the expansion dictionary. If expanding, this should be the expansion dictionary. In either case, the dictionary must begin on a page boundary, as the low order 12 bits of the address are assumed to be 0 when determining the address of the dictionary.

  If running in AR mode, set field CMPSC_SOURCEALET to the ALET of the necessary dictionary. Note that the input area is also accessed using this ALET. If not in AR mode, make sure that the field contains 0.
- In most cases, make sure that 3-bit field CMPSC_BITNUM in byte CMPSC_DICTADDR_BYTE3 is zero. This field has the following meaning:
  - If compressing, place the first compression symbol at this bit in the leftmost byte of the target operand. Normally this field should be set to 0 for the start of compression.
  - If expanding, expand beginning with the compression symbol that begins with this bit in the leftmost byte of the source operand. Normally this field should be set to the value used for the start of compression.
- Set word CMPSC_TARGETADDR to the address of the output area. For compression, the output area contains the compressed data; for expansion, it contains the expanded data.

  If running in AR mode, set field CMPSC_TARGETALET to the ALET of the output area. If not in AR mode, make sure that the field contains 0.
- Set word CMPSC_TARGETLEN to the length of the output area.

- Set word CMPSC_SOURCEADDR to the address of the input area. For compression, the input area contains the data to be compressed; for expansion, it contains the compressed data.

  If running in AR mode, set field CMPSC_SOURCEALET to the ALET of the input area. Note that the dictionary will also be accessed using this ALET. If not in AR mode, make sure that the field contains 0.
- Set word CMPSC_SOURCELEN to the length of the input area. For expansion, the length should be the difference between CMPSC_TARGETLEN at the completion of compression and CMPSC_TARGETLEN at the start of compression, increased by 1 if field CMPSC_BITNUM was nonzero upon completion of compression.
- Set word CMPSC_WORKAREAADDR to the address of a 192-byte work area for use by the CSRCMPSC macro. The work area should begin on a doubleword boundary. This area does not need to be provided and the field does not have to be set if your code has verified that the hardware CMPSC instruction is present. The program can do the verification by checking that bit CVTCMPSH in mapping macro CMSCVT is on.

When the CSRCMPSC service returns, it has updated the input CSRYCMPS area as follows:

- CMPSC_FLAGS is unchanged.
- CMPSC_DICTADDR is unchanged, but bits CMPSC_BITNUM in field CMPSC_DICTADDR_BYTE3 are set according to the last-processed compression symbol.
- CMPSC_TARGETADDR is increased by the number of output bytes processed.
- CMPSC_TARGETLEN is decreased by the number of output bytes processed.
- CMPSC_SOURCEADDR is increased by the number of input bytes processed.
- CMPSC_SOURCELEN is decreased by the number of input bytes processed.
- CMPSC_WORKAREA is unchanged.

The target/source address and length fields are updated analogously to the corresponding operands of the MVCL instruction, so that you can tell upon completion of the operation how much data was processed and where you might want to resume if you wanted to continue the operation.

Compression and Expansion Examples

The following is an example of compression.

```
          LA    13,SAVEAREA             Get address of save area
          LA    2,MYCBLOCK              Get address of parm
          USING CMPSC,2
          XC    CMPSC(CMPSC_LEN),CMPSC   Clear block
          OI    CMPSC_FLAGS_BYTE2,CMPSC_SYMSIZE_5 Set size
*                      Symbol size is 5+8. Dictionary has
*                      2**(5+8) entries
          L     3,DICTADDR
          ST    3,CMPSC_DICTADDR        Set dictionary address
          L     3,COMPADDR
          ST    3,CMPSC_TARGETADDR      Set compression area
          L     3,COMPLEN
          ST    3,CMPSC_TARGETLEN       Set compression length
          L     3,EXPADDR
          ST    3,CMPSC_SOURCEADDR      Set expansion area
          L     3,EXPLEN
          ST    3,CMPSC_SOURCELEN       Set expansion length
          LA    3,WORKAREA
          ST    3,CMPSC_WORKAREAADDR    Set workarea address
          CSRCMPSC CBLOCK=CMPSC
          DROP  2
  .
  .
          DS    0F        Align parameter on word boundary
MYCBLOCK  DS    (CMPSC_LEN)CL1    CBLOCK parameter
COMPADDR  DS    A         Output "To" (compression) area
COMPLEN   DS    F         Length of "To" area
EXPADDR   DS    A         Input "From" (expansion) area
EXPLEN    DS    F         Length of "From" area
DICTADDR  DS    A         Address of compression dictionary
          DS    0D        Doubleword align workarea
WORKAREA  DS    CL192     Work area
```

```
    SAVEAREA DS    CL144     Register save area
             CSRYCMPS ,
```

Note: The expansion dictionary must immediately follow the compression dictionary and both must be aligned on page boundaries.

The following is an example of expansion.

```
             LA    13,SAVEAREA              Get address of save area
             LA    2,MYCBLOCK               Get address of parm
             USING CMPSC,2
             XC    CMPSC(CMPSC_LEN),CMPSC   Clear block
             OI    CMPSC_FLAGS_BYTE2,CMPSC_SYMSIZE_5 Set size
    *                      Symbol size is 5+8. Dictionary has
    *                      2**(5+8) entries
             OI    CMPSC_FLAGS_BYTE2,CMPSC_EXPAND  Do expansion
             L     3,EDICTADDR
             ST    3,CMPSC_DICTADDR         Set dictionary address
             L     3,EXPADDR
             ST    3,CMPSC_TARGETADDR       Set expansion area
             L     3,EXPLEN
             ST    3,CMPSC_TARGETLEN        Set expansion length
             L     3,COMPADDR
             ST    3,CMPSC_SOURCEADDR       Set compression area
             L     3,COMPLEN
             ST    3,CMPSC_SOURCELEN        Set compression length
             LA    3,WORKAREA
             ST    3,CMPSC_WORKAREAADDR     Set workarea address
             CSRCMPSC CBLOCK=CMPSC
             DROP  2
      .
      .
             DS    0F        Align parameter on word boundary
    MYCBLOCK DS    (CMPSC_LEN)CL1    CBLOCK Parameter
    EXPADDR  DS    A         Output "To" (expansion) area
    EXPLEN   DS    F         Length of "To" area
    COMPADDR DS    A         Input "From" (compression) area
    COMPLEN  DS    F         Length of "From" area
    EDICTADDR DS   A         Address of expansion dictionary
             DS    0D        Doubleword align workarea
    WORKAREA DS    CL192     Work area
    SAVEAREA DS    CL144     Register save area
             CSRYCMPS ,
```

Note: The expansion dictionary must be aligned on a page boundary.

Suppose that you had compressed a large area but wanted to expand it back into a small area of 80-byte records. You might do the expansion as follows:

```
          LA   13,SAVEAREA              Get address of save area
          LA   2,MYCBLOCK
          USING CMPSC,2
          XC   CMPSC(CMPSC_LEN),CMPSC
          OI   CMPSC_FLAGS_BYTE2,CMPSC_SYMSIZE_1
          OI   CMPSC_FLAGS_BYTE2,CMPSC_EXPAND
          L     3,EDICTADDR             Address of expansion dictionary
          ST    3,CMPSC_DICTADDR        Set dictionary address
          L     3,EXPADDR
          ST    3,CMPSC_SOURCEADDR      Set compression area
          L     3,EXPLEN
          ST    3,CMPSC_SOURCELEN       Set compression length
          LA    3,WORKAREA
          ST    3,CMPSC_WORKAREAADDR    Set work area address
MORE      DS    0H                      Label to continue
*
* Your code to allocate an 80-byte output area would go here
*
          ST   x,CMPSC_TARGETADDR       Save target expansion area
          LA   3,80                     Set its length
          ST   3,CMPSC_TARGETLEN        Set expansion length
          CSRCMPSC CBLOCK=CMPSC         Expand
          C    15,=AL4(CMPSC_RETCODE_TARGET) Not done, target used up
          BE   MORE                     Continue with operation
          DROP 2
  .
  .
          DS   0F        Align parameter on word boundary
MYCBLOCK  DS   (CMPSC_LEN)CL1    CBLOCK Parameter
```

```
EXPADDR  DS   A         Input expansion area
EXPLEN   DS   F         Length of expansion area
EDICTADDR DS  A         Address of expansion dictionary
         DS   0D        Doubleword align work area
WORKAREA DS   CL192     Work area
SAVEAREA DS   CL144     Register save area
         CSRYCMPS ,     Get mapping and equates
```

Note that this code loops while the operation is not complete, allocating a new 80-byte output record. It does not have to update the CMPSC_BITNUM, CMPSC_SOURCEADDR, or CMPSC_SOURCELEN fields, because the service sets them up for continuation of the original operation.

If running in AR mode, the example would also have set the CMPSC_TARGETALET and CMPSC_SOURCEALET fields. The XC instruction zeroed those fields as needed when running in primary addressing mode.

Determining if the CSRCMPSC Macro Can Be Issued on a System

The following should be run to determine if your system contains the software or hardware necessary to run a CSRCMPSC macro:

```
* Check the CMS Level before testing bits in CVTFLAG2.
* Back level CMS releases fill the CVTFLAGS word with X'FFFF',
* since it was not a supported field before CMS Level 12.
*
         XR    R15,R15                 Clear a register
         ST    R15,RETCODE             Initialize return code
*
         LA    R1,QUERYCMD             Point to Query CMSLEVEL command
         CMSCALL PLIST=(1),ERROR=*     CMSCALL to execute command
         LTR   R15,R15                 Check the return code
         BZ    CHKLVL                  OK, continue
         ST    R15,RETCODE             Else, save bad RC
         LR    R3,R15                  Display error message
         APPLMSG TEXT='Q CMSLEVEL return code= &&1',           X
               SUB=(HEX,((R3),8))
         B     EXIT                    Go exit
*
CHKLVL   DS    0H
         ST    R1,QRESULT              Save result of query
         CLI   QRESULT+1,CMS12         Is this CMS Level 12?
         BNL   CHKCVT                  Equal or Higher? continue
         APPLMSG TEXT='CMS release is not CMS Level 12 or higher.'
         APPLMSG TEXT='Data Compression Services are NOT Supported.'
         LA    R15,111                 No, set bad RC
         ST    R15,RETCODE             Save bad RC
         B     EXIT                    Go exit
*
CHKCVT   DS    0H
         USING NUCON,0
         L     R8,ACMSCVT              Get address of CVT
         USING CMSCVT,R8
         TM    CVTFLAG2,CVTCMPSC       Is compression supported?
         BO    COMP_OK                 Yes, continue
*
         APPLMSG TEXT='Data Compression Services are NOT Supported.'
         LA    R15,111                 No, set bad RC
         ST    R15,RETCODE             Save bad RC
         B     CHK_HDWR                Continue checks
*
COMP_OK  DS    0H
         APPLMSG TEXT='Compression Services Supported.'
CHK_HDWR DS    0H
         TM    CVTFLAG2,CVTCMPSH       Is HW compression installed?
         BO    HDWR_OK                 Yes, continue
         APPLMSG TEXT='Machine does NOT have CMPSC HW compression.'
         B     EXIT                    Go exit
HDWR_OK  DS    0H
         APPLMSG TEXT='Machine has CMPSC hardware compression.'
*
* Return to caller
*
EXIT     DS    0H
         L     R15,RETCODE             Set return code
         ....
         BR    R14
```

```
*
* DATA AREAS
*
         DS    0D
RETCODE  DC    F'0'                    Save area for return code
QRESULT  DC    F'0'                    Result of query returned in R1
*
         DS    0D
QUERYCMD DC    CL8'QUERY'              Query CMSLEVEL command
         DC    CL8'CMSLEVEL'
         DC    8X'FF'
         LTORG
*
* Mapping macros
         NUCON
         CMSCVT
         CMSLEVEL
         REGEQU
```

High-Level Language Call

High-level languages like PL/I, Fortran, Cobol, and so forth can use the DMSCPR CSL call interface

```
Call DMSCSL,("DMSCPR  ",RETCODE,CBLOCK),VL
```

to compress or expand their data. For more information on Data Compression Services with CSL, see the DMSCPR CSL routine in the *z/VM: CMS Callable Services Reference*.

Compressing CMS Data

The following is a sample scenario you can follow to compress data using Data Compression Services. For more information on compressing and expanding your data, see Appendix L, "Data Compression Services," on page 609.

- Create a spec file to use as input to the CSRBDICV exec. The following is an example using TEXT SPECFILE.

```
**The following is with a 4K-entry dictionary.
**Provides 30.88% compression (output/input) for the source of
**Chapter 5 of the ESA/390 Principles of Operation (30.32% if all output
**bits are concatenated together).
**Optimization (change x under opt to opt) improves compression by 0.7%.
**results maxnodes maxlevels msglevel stepping prperiod dicts
  r       40000    60        3        f 7 2 7  1000     af asm
**colaps opt treedisp treehex treenode dupccs
  aam     x   x        h       n        x
**FLD col type dcenmen             INT  intspec
  FLD 1   sa   dce 4               INT  aeis 1 (40)
                                   INT  a12b3s (40)
  FLD end
```

  See the *z/VM: CMS Commands and Utilities Reference* for more information on the CSRBDICV EXEC and CSRCMPEV EXEC formats.

- To compress data, you will build both a compression and expansion dictionary. You must have a read/write disk set up as your A-disk, since your output files will appear on your A-disk. The source file that you want to compress using the created dictionaries can reside on any readable CMS disk.

  The following command will invoke the CSRBDICV EXEC to build dictionaries for the file CMSNUC MAP B:

```
csrbdicv 4 1 EB cmsnuc map b (text specfile a (build dictionary
```

  The compression and expansion dictionaries have now been built:

  sourcefilename ACDICT*sf* A (compression dictionary)
  sourcefilename AEDICT*sf* A (expansion dictionary)
  sourcefilename CEDICT*sf* A (abstract dictionary for CSRCMPEV)

For this example, the filetypes created would be: ACDICT41, AEDICT41, and CEDICT41, respectively. The *sourcefilename* would be CMSNUC.

- You can use the CSRCMPEV EXEC to compress and expand the data and report statistics. In this example, CSRCMPEV will use the abstract dictionary CMSNUC CEDICT41 A for compression and expansion.

  The following command invokes the CSRCMPEV EXEC to compress and expand the file CMSNUC MAP B using the *sourcefilename* dictionary built by the CSRBDICV EXEC from the previous example.

  ```
  csrcmpev 4 1 nhd cmsnuc map b cmsnuc (18550 18550 (1
  ```

  After you have created the dictionaries, rename them to *newfilename* ASSEMBLE A and assemble both files. Assemble the files by entering:

  ```
  hasm newfilename
  ```

  After being assembled, these dictionaries are in TEXT deck format. These TEXT deck files will be used by the application as the dictionaries.

Now that the dictionaries have been created, you can code a program which will call the Data Compression Services interface: CSRCMPSC for Assembler language or the DMSCPR CSL interface for a high-level language. Load the program and dictionaries into memory and then execute the program. Remember when you are compressing data the compression dictionary is loaded first, with the expansion dictionary immediately following. The program will pass the address of the compression dictionary to Data Compression Services. The service routine can then find both dictionaries by size offset. If the program wants to expand already compressed data, it will pass the address of the expansion dictionary and a flag bit indicating the expand option.

For compression, both dictionaries are required and they both must begin on a page boundary with the expansion dictionary immediately following the compression dictionary. For expansion, only the expansion dictionary is required and it must begin on a page boundary.

The following is a simplistic example of how this can be done using CMS LOAD and INCLUDE commands. The example assumes that the program and dictionaries are each 4096 bytes (1 memory page) long. These same actions can be executed wholly by the application program itself using storage obtained in either the primary address space or in a data space area for the dictionaries and Data Compression Services parameter list.

```
LOAD PGMNAME (ORIGIN 20000
INCLUDE CMPDICT (ORIGIN 21000
INCLUDE EXPDICT (ORIGIN 22000
START PGMNAME
```

Note: In the previous example, the dictionaries contain 512 (8 byte) dictionary entries. Each dictionary is 4096 bytes in total length (8 bytes * 512 = 4096). Programs using larger dictionaries with more entries will need to adjust the origins used accordingly.

Attention: Do not delete your TEXT and ASSEMBLE filetype dictionaries. You will not be able to expand your data if the TEXT deck files are deleted. It is recommended that the compression and expansion dictionaries be backed up along with any compressed vital data to ensure that the data can be completely restored.

Part 4. Connectivity Programming in CMS

In its simplest form, connectivity is the ability of one program to communicate with another program.The key questions to ask are:

- What kind of programs are they?
- Where are the programs located?
- How do the programs communicate with each other?

This part answers the preceding questions by describing:

- The environments in which the programs reside
- The different types of communications programs
- The communications programming terminology—SNA terms versus z/VM terms
- How your applications can use CPI Communications (also known as SAA communications interface) to communicate with each other.

Chapter 30. Introduction to Connectivity Programming in CMS

Application programs are typically written to communicate with one another because a user needs access to some kind of data. How these programs communicate depends on where they are located. The application programs can be located in any one of the following environments:

- Both programs are on the same z/VM system.
- The two programs are on different z/VM systems, but in the same collection. A collection is a group of z/VM systems logically connected together. We will discuss more about collections later in this chapter.
- One program is located in a z/VM system, and the other is located in a workstation on a ***local area network*** (***LAN***).
- The two programs are located in two different collections.
- One program is located in a z/VM system, and the other is located in a network defined by IBM's ***System Network Architecture (SNA)***.

A network is a group of two or more interconnected computing units that lets information be electronically transmitted from one computing unit to another. The information transmitted can range in size from a one-line transaction to a book-size online document. 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 Communications (APPC)***. APPC is also known as SNA ***LU 6.2***. z/VM implements the base set of APPC and several APPC option sets using ***Advanced Program-to-Program Communication/VM (APPC/VM)***.

z/VM provides two programming interfaces to APPC/VM:

- A low-level interface intended for programs written in assembler language.
- ***Common Programming Interface (CPI) Communications***, which is intended for programs written in REXX and high-level languages. CPI Communications (also known as SAA communications interface) is part of IBM's Systems Application Architecture (SAA), and it provides a standard set of routines and parameters for calling these routines. Programs using this interface can be more easily transported across other IBM environments. Refer to the *Common Programming Interface Communications Reference (https://publibfp.dhe.ibm.com/epubs/pdf/c2643999.pdf)* for details about CPI Communications routines.

The following communications programming interface is also available in z/VM:

- The Inter-User Communication Vehicle (IUCV), which is part of z/VM. IUCV is for communications between two programs on the same z/VM system, and it also allows a program to communicate with a CP system service. For complete information on IUCV, see *z/VM: CP Programming Services*.

Types of Communications Programs

Typically, one application program requests a resource from another application program. This resource might be data, a file, or access to a device. The application program that requests resources is known as a ***user program***. User programs run in ***requester*** virtual machines.

A virtual machine that provides access to a resource is known as the ***server*** virtual machine. Some examples of server virtual machines are virtual machines that:

- Contain a database manager
- Contain a file server
- Manage a high-function printer.

The program that actually controls a resource is called a ***resource manager program*** in z/VM. Resource manager programs run in server virtual machines.

We will talk about resources and resource managers in more detail in Chapter 32, “Program-to-Program Communications,” on page 479.

Figure 74 on page 460 shows a CMS user program, running in a requester virtual machine, communicating with a CMS resource manager program that is running in a server virtual machine, both in a single z/VM system.

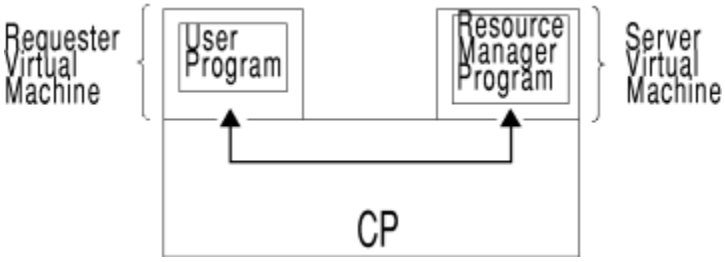


Figure 74. Communications between a User Program and a Resource Manager

How the Programming Interfaces Work Together

Figure 75 on page 461 shows how a user program can use either the CPI Communications interface or the APPC/VM assembler language programming interface to use the APPC/VM services that support program-to-program communication. Using these services, the APPC/VM (user) program can communicate with other programs on:

- The same z/VM system
- Different z/VM systems
- Non-z/VM systems.

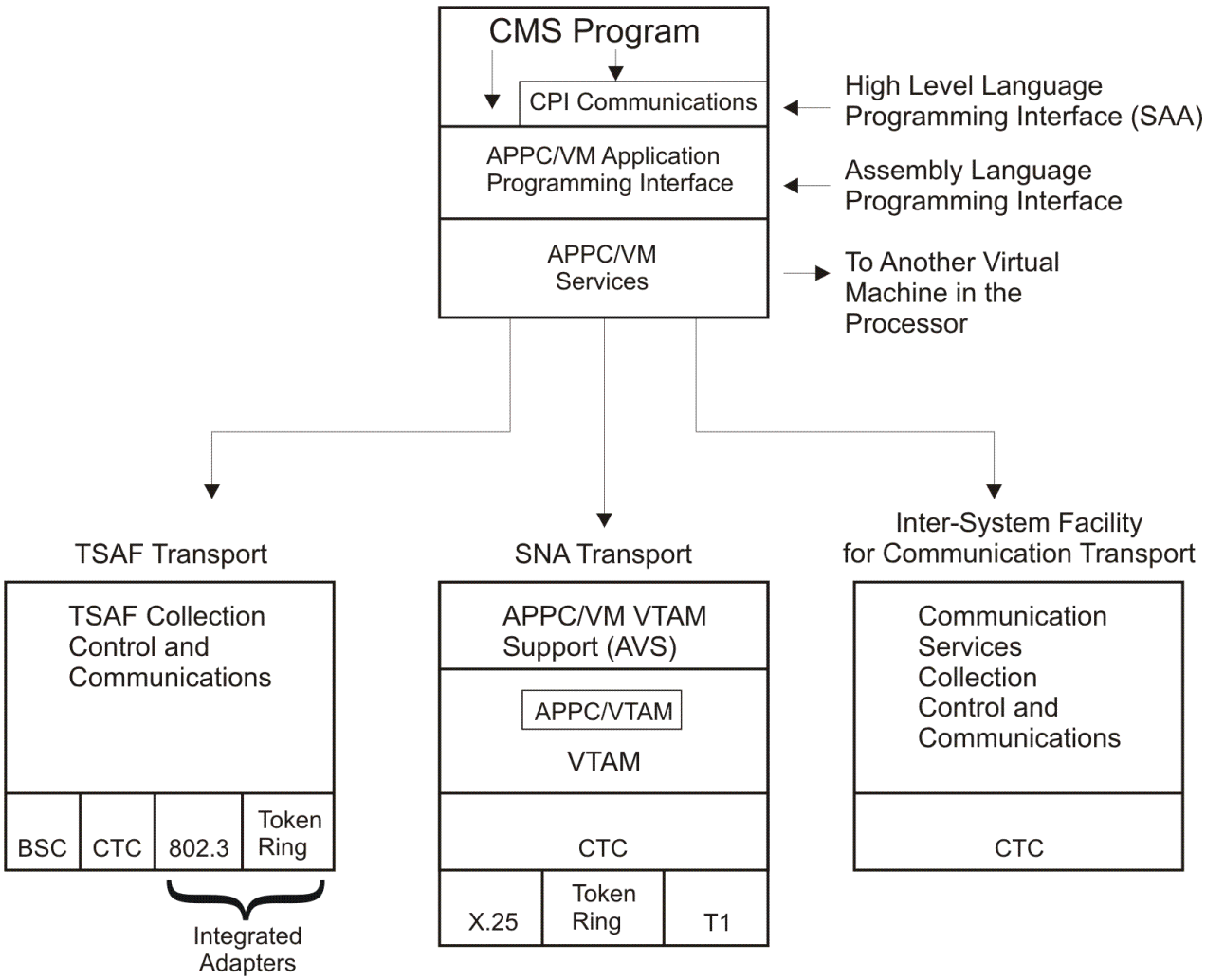


Figure 75. APPC/VM Programming Interfaces

Understanding the Scope of APPC/VM Communications

User programs and resource manager programs communicate in z/VM through APPC/VM. The two partners start communicating when the user program requests a service that the resource manager program controls. The figures in this section show how these communications work. In these figures, application program refers to any program that uses APPC/VM, regardless of its programming interface.

Communication within a Single z/VM System

A program running in your virtual machine can communicate with and request services from a program running in another virtual machine in the same system. The Conversational Monitor System (CMS) and Control Program (CP) components of z/VM handle communications between the virtual machines. Figure 76 on page 462 shows how data, being exchanged between programs running in two virtual machines in the same system, passes through CMS and CP.

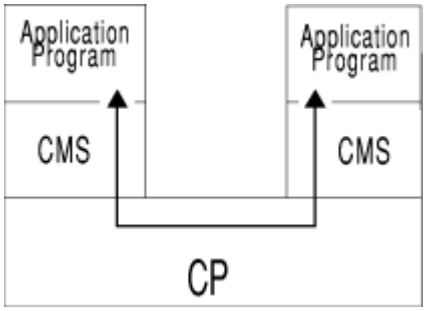


Figure 76. Communication within One z/VM System

For example, suppose you have a program running in your virtual machine that needs to access data in a database. In the other virtual machine, a database manager program (resource manager program) is running. When your program asks for information from the database, APPC/VM passes this request to the database manager program; the database manager program then gets the requested data and sends the data back to your program using APPC/VM.

Communication within a TSAF Collection of z/VM Systems

Organizations with more than one z/VM system might need to share resources (such as a database) among users on these z/VM systems. The Transparent Services Access Facility (TSAF) virtual machine provides interprocessor communications services for a ***TSAF collection*** of up to eight z/VM systems. (A TSAF collection can also consist of one z/VM system which does not require the TSAF virtual machine.) Programs, through APPC/VM, communicate with programs in other virtual machines located throughout the TSAF collection of z/VM systems. To support this type of communication, z/VM's TSAF component is required along with its CMS and CP components. In Figure 77 on page 462, the TSAF virtual machine (running on CMS) connects the two z/VM systems so that programs running in virtual machines on different systems can communicate with APPC/VM.

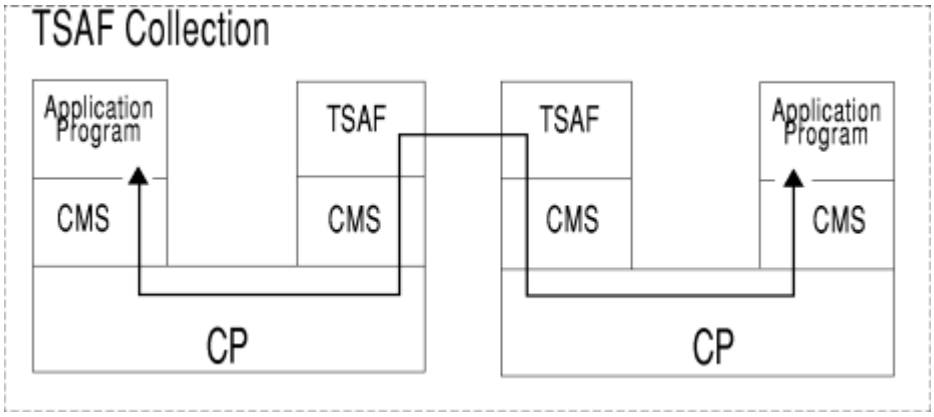


Figure 77. Communication within a TSAF Collection

As with the previous example, suppose your program requests access to data in a database. In this case, however, your program and the database manager program run in virtual machines in different systems. Because the TSAF virtual machine provides a transparent path to the other system, your program can communicate with the database manager program without knowing that the database is located on another system. As in the single z/VM system case, your program's data request passes to the database manager program; the database manager program then gets the requested data and sends it back to your program.

Communication Outside Your z/VM System, TSAF, or CS Collection

Your organization may need to share resources with another system or with systems that are not part of the same TSAF or CS collection. These systems may or may not be z/VM systems—and they could even be non-IBM systems. The Advanced Communications Function for the Virtual Telecommunications Access

Method (ACF/VTAM*, or VTAM for short), Group Control System (GCS), and APPC/VM VTAM Support (AVS) provide these types of communications services.

- ***ACF/VTAM*** controls telecommunications activity and interprocessor communications in an SNA network.
- ***GCS*** manages subsystems that support an SNA network and provides an interface between these subsystems and CP. ACF/VTAM runs in a GCS virtual machine on z/VM. GCS is a z/VM component.
- ***AVS*** provides the interface between ACF/VTAM and APPC/VM. AVS is also a z/VM component.

With AVS and ACF/VTAM, a program in a TSAF collection, through APPC/VM, can communicate with:

- Other programs residing in z/VM systems on different TSAF or CS collections or within an SNA network.
- Programs on non-z/VM systems in the SNA network.

AVS and VTAM support communications between a TSAF or CS collection and systems in the SNA network because they provide the SNA network with a view of the TSAF or CS collection. To support this type of communication, VTAM and AVS (as well as GCS) are required in a TSAF or CS collection along with CMS and CP.

Communication between a z/VM and Non-z/VM System

In Figure 78 on page 463, AVS and VTAM connect the TSAF collection (made up of two z/VM systems) to a non-z/VM system in the SNA network. AVS translates information between APPC/VM and APPC/VTAM (the VTAM implementation of APPC). VTAM provides the path between the TSAF collection and the system in the SNA network.

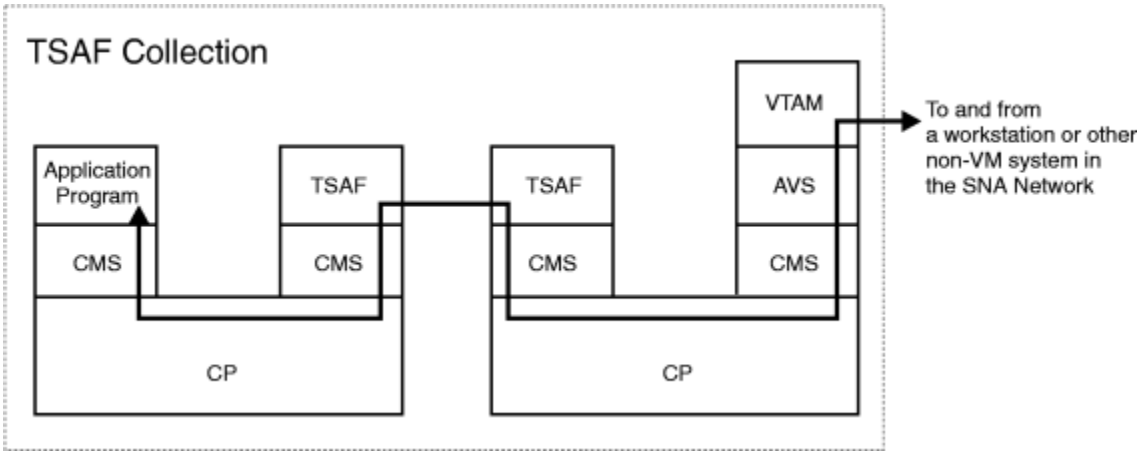


Figure 78. Communication between a TSAF Collection and an SNA Network

Suppose you have a program, running in a non-z/VM system located in the SNA network, that requests data from a database. Your request is made using APPC. The database manager program is located in a system in the TSAF collection. Your request is routed to the TSAF collection over a path through the SNA network between your workstation and VTAM in the TSAF collection. VTAM routes your request to AVS which translates the request from APPC/VTAM to APPC/VM. TSAF then routes the connection to the database manager program through the path established between TSAF virtual machines because AVS and the database manager are on different systems in the collection. The database manager program gets the data and then sends it back to your program.

CPI Communications programs on workstations and APPC/VM programs on VM/ESA systems in a CS collection can communicate with APPC programs located in an SNA network. The AVS component and VTAM provide a path between VM/ESA systems in the CS collections and the SNA network. ISFC, AVS, and VTAM provide a transparent connection between the CPI Communications or APPC/VM program in the CS collection and the APPC program in the SNA network. Using ISFC, AVS, and VTAM, programs in the CS collection can access wide area network resources and programs in this network can communicate with resources in the CS collection.

Communication between Two TSAF Collections

Figure 79 on page 464 illustrates an example that is similar to Figure 78 on page 463, except this time AVS and VTAM connect two TSAF collections (each made up of one z/VM system) in the SNA network.

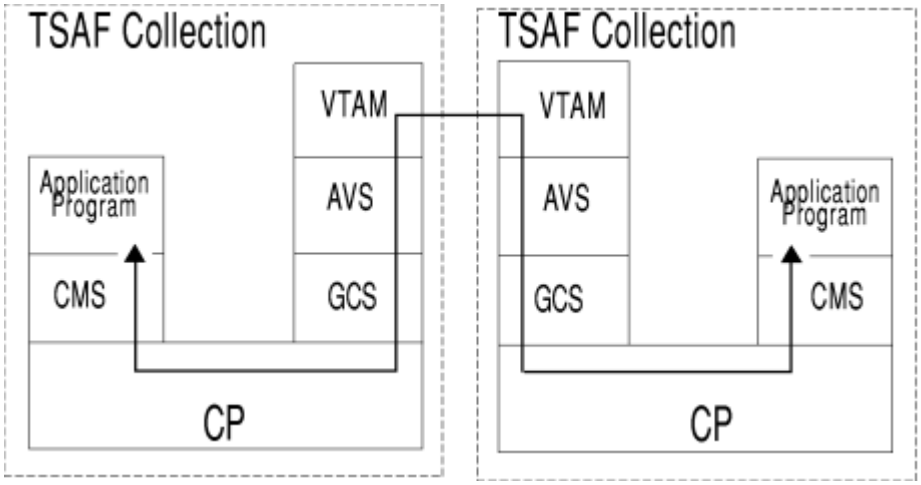


Figure 79. Communication between Two TSAF Collections

For example, suppose you have a program running in a TSAF collection, and that TSAF collection is part of an SNA network. Your program requests data, using APPC/VM, from a database. The database manager program is located in a different TSAF collection in the SNA network. When your program requests data from the database, the request goes to AVS, where it is transformed from APPC/VM to APPC/VTAM. The request then passes over the path established by VTAM to the second TSAF collection. AVS in the second collection translates the request from APPC/VTAM to APPC/VM. TSAF then passes the request to the database manager program. (The request is not routed through TSAF virtual machines, because AVS is located in the same system as the database manager.) The database manager program gets the requested data and then, through APPC/VM, sends it back to your program.

Summarizing z/VM Program-to-Program Communication

As described in this chapter and summarized in Figure 80 on page 465, the APPC/VM programming interface lets your APPC/VM programs in a VM system communicate with APPC programs located in:

- Same z/VM system
- Different z/VM system in the same TSAF collection
- z/VM system in an SNA network that has AVS and VTAM running
- z/VM system in a different TSAF collection that has AVS and VTAM running
- Non-z/VM system in an SNA network that supports the APPC protocol
- Workstation in an SNA network that supports the APPC protocol
- Non-IBM system in an SNA network that supports the APPC protocol.

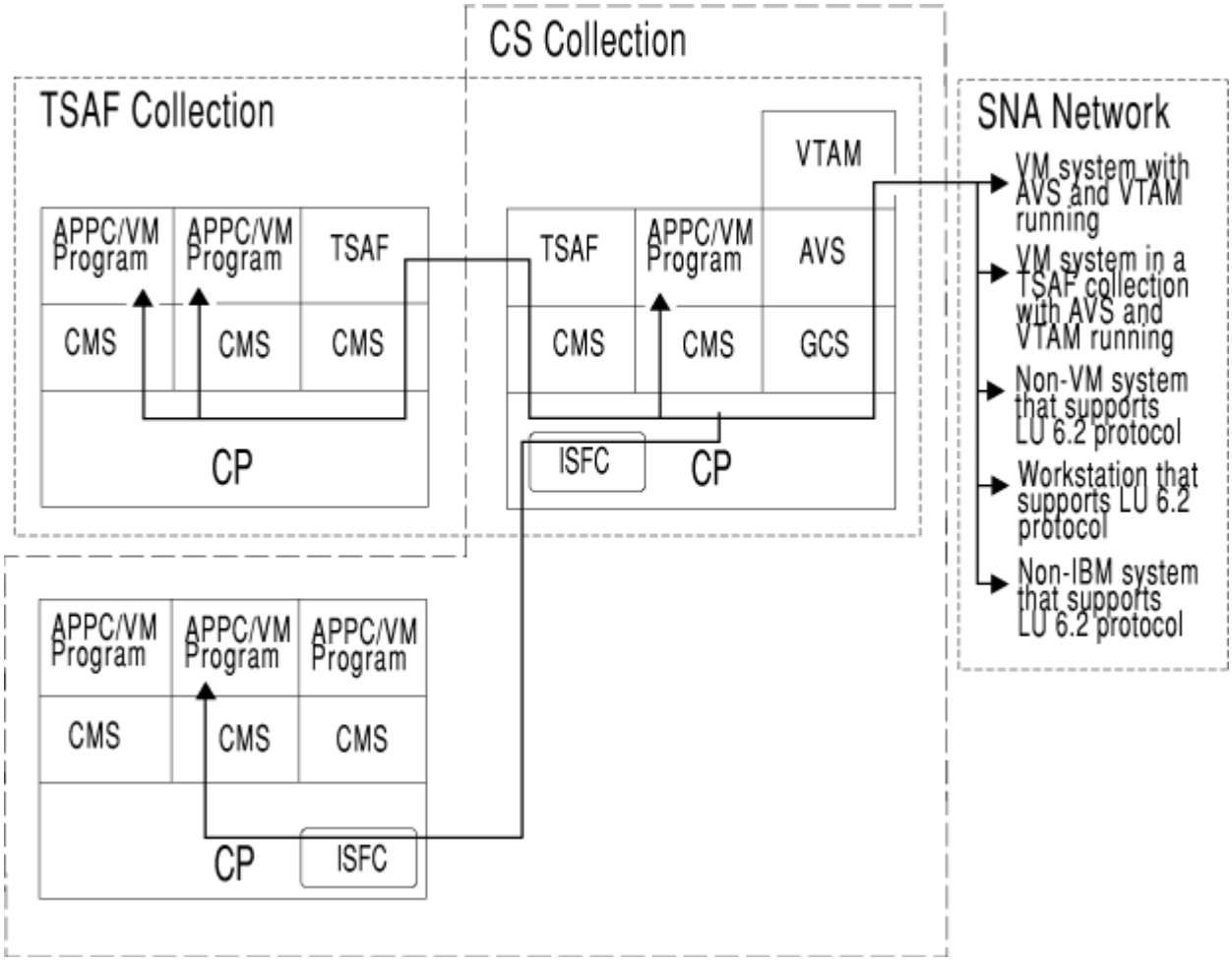


Figure 80. Summary of z/VM Connectivity

Chapter 31. Understanding Communications Programming Terminology

This chapter describes the terminology used with the APPC communications supported by TSAF, AVS, and ISFC. This terminology includes SNA communication terms and VM communication terms. See "Systems Network Architecture Terminology" on page 467 for the definitions of SNA terms. VM terms are defined under "VM Terminology" on page 470.

Systems Network Architecture Terminology

The following SNA terms are briefly described in this section. See the *Systems Network Architecture Transaction Programmer's Reference Manual for LU Type 6.2* for detailed descriptions of SNA terms and concepts.

- SNA network
- Logical unit
- Session
- Transaction program
- Conversation
- Mode name
- Session limit
- Contention
- Session security
- Conversation security
- Negotiation.

What Is an SNA Network?

A network is a group of two or more interconnected computing units that lets information be electronically sent from one computing unit to another. The information sent can range in size from a one-line transaction to a book-size online document. An ***SNA network***:

- Enables the transfer of data between end users (typically, terminal operators and application programs), and
- Provides protocols for controlling the resources of any specific network configuration.

What Is a Logical Unit?

The SNA network consists of physical processors, called ***nodes***, which are connected by physical data links. The SNA network also consists of logical processors, called ***logical units (LUs)***. An LU lets a user gain access to network resources (such as programs) and communicate with other users. LUs provide protocols that let users communicate with each other.

LU 6.2 is a particular type of SNA logical unit. LU 6.2 provides a connection between its users and network resources (often called transaction programs). The protocol that LU 6.2 provides is called Advanced Program-to-Program Communications (APPC).

What Is a Session?

In SNA, a ***session*** is a logical connection between LUs. Sessions can be compared to phone lines through which data flows between the LUs. An LU 6.2 can support more than one session concurrently with the

same partner LU. Such sessions are called ***parallel sessions***. An LU 6.2 can support sessions with multiple LUs.

What Is a Transaction Program?

A ***transaction program*** is an application program that helps users access resources in a network. A transaction program uses the services provided by the LU to communicate with other transaction programs by issuing transaction program verbs (APPC functions). A transaction program depends on program-to-program communications with another transaction program for some or all of its processing.

What Is a Conversation?

While LUs are connected by sessions, transaction programs are connected by ***conversations***. Just as a session is a logical connection between the LUs, so is the conversation a logical connection between two transaction programs. LU 6.2 treats a session as a reusable connection between two LUs. One session can support only one conversation at a time, but one session can support many conversations in sequence. Because sessions are reused by multiple conversations, a session is a long-lived connection compared to a conversation.

To establish a conversation, a transaction program specifies the name of the remote LU, the name of the transaction program, the session mode name (that identifies certain session characteristics), and security parameters. The LU name specified by the transaction program is a network addressable unit that routes the connection within the SNA network.

If a session with the characteristics of the specified mode name exists between the two LUs and the session is not being used for another conversation, the LUs assign that session to the new conversation for its use. If a session is not available, the LU starts a new session using the specified mode name. This new session is then used for the conversation.

After the conversation is initiated, the two transaction programs use LU 6.2 verb functions to send and receive data as necessary to accomplish the transaction. When the transaction is finished, one transaction program ends the conversation. That session is now available for another conversation between transaction programs using the session's LUs as entry points into the SNA network.

What Is a Mode Name?

LU 6.2 associates each session with a set of characteristics called a ***mode name***. Mode names define characteristics such as pacing levels and class of service. Examples of pacing levels and classes of service are secure, ASCII data, satellite communication, high speed, batch, and interactive. When transaction programs request a conversation, they cannot specify which session to use for the conversation, but they can specify what the characteristics of the session are. Transaction programs specify these characteristics by the mode name. Mode names are assigned by the system administrator.

All sessions between a pair of LUs that have the same mode name are called ***session groups***. These sessions form a group that can be treated as a pool of sessions sharing the mode name characteristics. Programs can control the size of the pool, but the LU is responsible for actually handling the individual sessions that make it up.

LUs can define several different session groups for sessions with another LU. Thus, an LU could have a FILESERV mode name defined for sessions with a partner LU used by file-server programs, and other mode names such as INTERACT defined for sessions with the same LU used for database queries. In this example, FILESERV could denote sessions with a large request size, which would aid bulk transmission of data.

What Is a Session Limit?

LU 6.2 does not allow an unlimited number of sessions between a pair of parallel-session capable LUs. Limits are imposed on the number of total sessions LUs may have between them. These limits are called ***session limits***. Because each mode name has its own session limits, LU pairs have multiple session limits.

LUs may be limited, for example, to five sessions with another LU using a mode name of HISPEED, while being able to have 10 sessions with the same LU using a mode name of SECURE.

Session limits can be dynamically defined while the programs are executing. LU 6.2 defines Change Number of Session (CNOS) verbs that can change session limits. The session limit defined by the CNOS verbs limits the number of conversations that can be active. If there are 10 sessions, then there can only be 10 concurrent conversations active.

What Is Contention?

It is possible for two LUs to attempt to allocate a conversation over the same session at the same time. This situation is called ***contention***. It is resolved by designating one of the LUs the ***contention winner*** for the use of the session. The other LU is the ***contention loser***.

A contention winner can allocate the session without informing the contention loser. A contention loser can request use of the session from the contention winner. For LU 6.2 programs, the LU is responsible for requesting use of sessions on the contention loser side and granting or denying it on the contention winner side. Programs need not be concerned with this process.

Generally, parallel session-capable LUs can divide the contention winner role between them for their sessions. The number of winners is divided based on which LU will typically start a conversation. For example, if two LUs have 10 parallel sessions and if the conversations will be equally started by both LUs, one LU may be designated the contention winner for five sessions and the other LU the contention winner for five sessions. The number of contention winners can be defined at system definition time or dynamically. A CNOS verb sets the contention winner and contention loser values. When you enter a CNOS verb, you may not want to set the contention loser and winner values equal. For example, if a workstation is attached to a host, the workstation would define a large number of contention winners and a small number of contention losers because it will initiate more conversations to connect to the host than the host will initiate to connect to the workstation.

What Is Session Security?

Session security or session level LU-LU verification verifies the identity of each LU to its session partner LU during session activation. The same LU password specification must be defined at both LUs. There are two types of password specification:

- NONE that indicates no LU-LU password is to be defined.
- NAME that specifies the LU-LU password.

What Is Conversation Security?

For each conversation, ***conversation security*** information is sent to the target LU and the target transaction program. The target LU receives an access security field that indicates the type of security being used, an access security user ID that uniquely identifies the source (requesting) program, and an access security password. The target LU verifies the user ID and password. The LU then uses the verified user ID to determine if the source transaction program can connect to the target transaction program. The target transaction program receives the access security type and the access security user ID.

The extent to which a target transaction program uses the access user ID to determine if the requester can connect can vary. Some transaction programs, like a public bulletin board, choose not to use the access user ID. They let any transaction program access the information. Other transaction programs, like a database manager, can use the access user ID to determine which data can be accessed by the source transaction program.

The source transaction program specifies in its allocation request what type of access security information will be sent to the target LU and target program. The three types of access security defined by APPC are:

- SECURITY(PGM)
- SECURITY(SAME)

- SECURITY(NONE).

Specifying SECURITY(PGM) indicates that the source transaction program is sending an access security user ID and password in its allocation request. The target LU verifies the access security user ID and password and determines if the source program is authorized to connect to the target transaction program. The target transaction program receives the access security user ID and can use it to determine if the source program can access its resources.

Specifying SECURITY(SAME) indicates that the source transaction program is not sending an access security user ID or password in its allocation request. The access security user ID that was used to invoke the source transaction program is sent to the target LU and transaction program. The target LU and target transaction program determine if the source program should be allowed to connect.

In APPC/VM, the requesting program that specifies SECURITY(SAME) does not specify the access user ID in the allocation request. The logon user ID that was used to invoke the requesting program when the user logged on is sent by CP to the target transaction program in the connection request information. The target transaction program can then determine who is attempting to communicate and can decide if the requester should be allowed to communicate. VM's Shared File System (SFS) is an example of a transaction program that uses the access user ID (logon user ID) to determine which users are authorized to use certain files.

SECURITY(SAME) should be used when a program requests services for another program. For example, the source program (B) was invoked by a transaction program (A) which specified SECURITY(PGM). B in turn requests the services of another transaction program (C) to satisfy the request of A. When B requests access to C, it specifies SECURITY(SAME) indicating that C will receive the access security user ID that was used to invoke B (A's user ID) not B's access security user ID. The source program (B) is called an ***intermediate server***, because it requests services from the target (C) for A.

Specifying SECURITY(NONE) indicates that the source transaction program is not sending any access security information in its allocation request. The target LU and transaction program will not receive an access security user ID. The remote LU cannot determine if the source transaction program is authorized to connect to the target transaction program. The target transaction program cannot determine which transaction program is requesting a connection and is written to allow any transaction program to connect. A public bulletin board is an example of a program that does not need to know which users access it and that would allow any user access. If a conversation is allocated with SECURITY(NONE), a workstation not part of the LU where the bulletin board resides could access the bulletin board.

What Is Negotiation?

To coordinate activities in an SNA network, the LU 6.2 protocol lets LUs negotiate values. For example, one LU has issued a CNOS verb that indicates it wants a session limit of 10,000 sessions between itself and its partner LU. The partner LU may only want to have 5,000 sessions between itself and the other LU. The partner LU can negotiate with the other LU to decrease the number of sessions down to 5,000. The contention winner and contention loser values defined on the CNOS verb and the access security levels can also be negotiated.

VM Terminology

The following VM connectivity terms are briefly described here. These terms are described in more detail in other sections of this book.

- TSAF collection
- CS collection
- Domain and domain controller
- VM resource
  - Local
  - Global
  - System

  - Private
- Communications partners
  - Resource manager
  - User program
- AVS Gateways
  - Global gateway
  - Private gateway
- System gateway.

What Is a TSAF Collection?

A ***TSAF collection*** is a group of up to eight interconnected z/VM systems each of which has a TSAF virtual machine installed and running.

A TSAF collection has the following properties:

- Automatic formation

  Systems can dynamically join and leave the collection.
- Automatic resource identification

  VM resources (transaction programs) can dynamically identify themselves within the TSAF collection without manual intervention.
- Transparent access to VM resources

  VM resources can be accessed by APPC/VM programs within the collection without regard to the resource's location.
- Single name space for global resources and user IDs

  Global resources are known throughout the collection and their names are unique within a collection. User IDs uniquely identify a particular user.

For programs in a TSAF collection to communicate, a logical connection must be established between the programs. Within a single VM system, CP provides an APPC/VM path that logically connects two programs. Within a TSAF collection, CP provides an APPC/VM path that connects each program with the TSAF virtual machine on its system. The TSAF virtual machines provide a logical APPC/VM path (a communications link) between the two systems, thus letting the two programs communicate.

What Is a CS Collection?

A ***Communication Services (CS) collection*** is a group of interconnected domains consisting of VM systems using ISFC to communicate. A CS collection has the following characteristics:

- Automatic formation

  Systems can dynamically join and leave the collection.
- Automatic resource identification

  Resources can dynamically identify themselves within the CS collection without manual intervention.
- Transparent access to resources

  Resources in the CS collection can be accessed by APPC/VM programs running on z/VM systems using ISFC in the collection.
- Single name space for global resources and user IDs

  Global resources are known throughout the CS collection and their names are unique within the collection. A user ID uniquely identifies a particular user.

What Is a Domain?

A ***domain*** consists of users and a domain controller. A group of interconnected domains form a CS collection. In a z/VM system running ISFC, CP acts as the domain controller for all of the users defined in the directory of that system and authorized to use APPC/VM communications.

Transaction programs can use ISFC to access resources, manage resources, and allow users to access shared resources. Transaction programs can reside in virtual machines on a VM system. Users of these transaction programs must sign-on to a domain to be able to access or manage resources. This sign-on ensures a user is authorized to use resources in the CS collection.

What Is a VM Resource?

A ***VM resource*** is a program, a data file, a specific set of files, a device, or any other entity or set of entities that you might want to identify for use in application program processing. A VM resource is identified by a VM resource name. A VM resource maps to a transaction program, and a resource name maps to a transaction program name.

A single program can be represented by one or more resource names. For example, a database program that manages two databases, DB1 and DB2, could be known by the resource name DB1 for requests to database DB1. However, the same program could be known by the resource name DB2 for requests to database DB2. Resources are managed by resource managers that run in server virtual machines (see "What Are Communications Partners?" on page 474).

A resource can be located on the local system or on any other system within the TSAF or CS collection. There are four types of resources in a TSAF or CS collection:

- Local
- Global
- System
- Private.

The following sections briefly describe the features of the four different types of resources. See *z/VM: Connectivity* for a detailed description of the different resource types.

What Are Local and Global Resources?

A ***local resource*** is known only to the local system, and a ***global resource*** is known to all systems in the TSAF or CS collection. Only authorized users on the local system can access local resources. When a resource is local, the names of the resources only need to be unique within the local system and not within the TSAF or CS collection. Resources (for example, a printer) that should be limited to the users of one system should be defined as a local resource to that system.

Authorized users in the TSAF or CS collection or the SNA network can access global resources. Each global resource name must be unique within the TSAF or CS collection in which it resides. Resources (for example, databases) that contain dynamic information needed by users in the TSAF or CS collection should be defined as global.

Global and local server virtual machines are explicitly logged on and the resource managers are explicitly invoked. Therefore, the resource managers are always ready for requests.

If a local and global resource are defined with the same name, the resources are accessed as follows:

- When a local user on the local VM system requests to communicate with the resource, CP routes the user to the local resource. TSAF or ISFC routes the local user to a global resource only if a local resource by that name does not exist.
- When a remote user on another VM system in the TSAF or CS collection requests to communicate with the resource, TSAF or ISFC routes the user to the global resource, even if a local resource with the same name also exists on the target system.

For example, there are two printers, where one is defined as a local resource on the local system and the other is defined as a global resource. Both resources are identified as PRINTER. When the local user requests to communicate with the local resource PRINTER and the local resource is unavailable, the local user will be routed to the global resource PRINTER.

What Are System Resources?

A ***system resource*** is known only to the z/VM system where it is located but is remotely accessible from other systems. A system resource name only needs to be unique to that system. Any authorized user in the TSAF or CS collection or the SNA network connected to the system on which the system resource resides can access the system resource. From a SNA system, they are accessed by an AVS global gateway. Note that AVS must be running on the same VM system as the target system resource. From a requester on the same VM system, they are accessed the same as a local or global resource on that system. From a TSAF or CS collection, system resources are accessible through the system gateway of the system on which the system resource resides. This is done using an LU name of SYSGATE, where SYSGATE is the system gateway of the target system, and a TPN of the name of the system resource. See "What Is a System Gateway?" on page 476 for more information. See Table 69 on page 489 for identifying the target of a connection request.

The server virtual machine must be authorized to manage a system resource. This authorization is the same that is required for a global resource manager.

Like local and global resource server virtual machines, the system resource server virtual machine needs to be logged on and the resource manager needs to be invoked before requests for a connection to that resource can be completed successfully.

What Are Private Resources?

A ***private resource*** is known only to the virtual machine in which it is located. It is not identified to CP, and not explicitly known throughout the TSAF or CS collection. In the private resource server virtual machine, there is a special NAMES file that lists the private resources and authorized users for each resource. Each private resource name within the TSAF or CS collection only needs to be unique within the virtual machine in which it resides. Any authorized users in the TSAF or CS collection or the SNA network can access private resources.

Resources (for example, a plotter) that are not frequently used should be defined as private. Resources that need to be limited to a single user should also be defined as private. For example, a user working on a workstation uses a program to access files in their virtual machine. These files would be defined as private resources and the workstation user would be the only authorized user of the resources. A resource that needs to be limited to a single user (like the previous example) or to a group of users (for example, a department) should be defined as private.

If a system administrator wants to control access to a resource rather than having the resource manager program control access, the resource should be defined as private. The system administrator can create a protected batch-like environment using private resources. The administrator can authorize the server virtual machine to issue privileged instructions. The administrator can also identify which programs run in the server virtual machine and limit which users can run certain programs.

Unlike local and global resource server virtual machines, the private resource server virtual machine does not need to be logged on and the resource manager does not need to be invoked when a program requests a connection. If the private resource server virtual machine is not logged on and its CP directory entry contains an IPL CMS statement, CP will automatically log it on and invoke the private resource manager.

In a CS or TSAF collection, private resource support has been enhanced with the system gateway. Using the system gateway in these environments, the same private server may be defined on all systems in a TSAF or CS collection. The system gateway may be used to selectively choose the desired private server. In this case the LU name would be SYSGATE USERID where SYSGATE is the system gateway name of the target system and USERID is the user ID of the private server.

Resource Interrelationships

Here are some resource interrelationships:

- A VM system can have local and global resources with the same name, and can also have local and system resources with the same name. In these cases, user requests originating on the same VM system for access to the resource will be given to the local resource. The global/system resources can only be accessed from a remote system.
- A VM system cannot have global and system resources with the same name.
- A TSAF or CS collection enforces unique global resource names within the collection.
- A TSAF collection may have a global resource defined at one system and also have system resources with the same name defined on other systems within the collection (one instance per system). However, in a CS collection, a system/global resource defined at one system precludes the definition of a global/system resource with the same name at any other node within the collection.
- A TSAF or CS collection can have system resources with the same name defined on multiple systems (one instance per system). Selective access to these resources can be made through the system gateway of the desired system.
- A single system can have a total of 500 global resources and gateways identified. With the combination of 500 global resources plus gateways, the potential exists for another 65,535 local resources and 65,535 system resources.

What Are Communications Partners?

Communications partners are transaction programs that communicate when one of the programs requests the services of the other transaction program. The transaction program that requests services is known as a ***requester*** or ***user program***. The service requested is access to a resource. The transaction program that provides this service or manages this resource is known as the ***server*** or ***resource manager program***. The user program is the communications partner of the resource manager program, and the resource manager program is the communications partner of the user program.

User programs and resource managers are written using the APPC/VM programming interface. Therefore, programs running in two virtual machines in a TSAF or CS collection can communicate through the APPC/VM programming interface. User programs or resource managers running in virtual machines in the TSAF or CS collection can communicate through the APPC/VM programming interface with APPC programs in the SNA network.

What Is a Resource Manager?

A ***resource manager*** is a program or set of programs executing in a virtual machine and managing access to one or more VM resources. A resource manager (a transaction program) gets requests from the user program (another transaction program) to access resources owned by the resource manager.

A resource manager runs in a server virtual machine. You can add entries to the server virtual machine's CP directory entry to authorize requester virtual machines to connect to the local, global, or private resource. You can also add entries to the private resource server virtual machine's special NAMES file to authorize requester virtual machines to connect to private resources. For more information on this special NAMES file and how to set up server virtual machines for different types of resources, see the *z/VM: Connectivity*.

Some examples of resource managers are:

- A database manager
- A file server that manages a set of files
- A virtual machine that manages a high-function printer.

What Is a User Program?

A ***user program*** is a program that runs in a requester virtual machine and depends on program-to-program communications with a resource manager for some or all of its processing. A user program starts a conversation with the other transaction program (a resource manager) when it requests a connection to a resource managed by the other transaction program. See *z/VM: Connectivity* for a description how to set up the requester virtual machine.

What Is an AVS Gateway?

Programs in the SNA network view a TSAF or CS collection as one or more LUs. The LUs defined to represent the TSAF or CS collection to VTAM are identified as AVS ***gateways*** in VM. Gateways are communication servers. Defining a gateway causes AVS to:

- Let the rest of the SNA network access a defined set of resources, the TSAF or CS collection as the LU whose name is the same as the AVS gateway's name.
- Communicate with the remote LUs in the SNA network for the APPC/VM programs in the collection as if these programs resided at the LU whose name is the same as the AVS gateway's name.

AVS gateways are managed by the AVS virtual machine. Gateways can be dynamically added or deleted. ISFC and the TSAF virtual machine keep up-to-date lists of all gateways in the collection. Each gateway name must be unique within the TSAF or CS collection where the AVS virtual machine resides.

AVS and VTAM provide services for any two programs using the APPC protocol to communicate with each other. One program, which uses APPC/VM, is located in the TSAF or CS collection. The other program is outside the TSAF or CS collection. So that the programs can communicate, AVS establishes a logical connection between the APPC/VM program in the TSAF or CS collection and the APPC program in the SNA network so that these programs can communicate. The TSAF virtual machine provides any necessary connections between the collection's VM systems. CP provides the connection between the APPC/VM program and AVS. ACF/VTAM provides the session between AVS and the remote LU.

Two types of gateways provide access to resources in the TSAF or CS collection: global gateways provide access to global and system resources; private gateways provide access to private resources. A global resource is known to all the systems in a TSAF or CS collection and a private resource is known only to a virtual machine. Therefore, a system can have both a global and private resource with the same name. Requesting a connection to a resource through a specific gateway type lets VM know which type of resource to search for in satisfying the request. For example, a request to connect to a resource through a private gateway lets VM know to look for a private resource. Global and private gateways are defined by the AVS command AGW ACTIVATE GATEWAY (described in *z/VM: Connectivity*). The gateway name is really the name of an LU for use in communicating over a SNA network.

What Is an AVS Global Gateway?

APPC programs in the SNA network use AVS ***global gateways*** to access global resources that reside in the TSAF or CS collection. Global resource managers use the global gateways to access APPC programs in the network. An SFS file pool is an example of a resource that could be identified as a global resource in a TSAF or CS collection. Users in a VM system located in the SNA network would access this file pool through the global gateway defined in the TSAF collection. Global gateways are also used to access system resources, but the resource must reside on the same system that the global gateway is identified.

What Is an AVS Private Gateway?

APPC programs in the SNA network use AVS ***private gateways*** to connect to private resources located in the TSAF or CS collection. Private gateways can be defined as either dedicated or nondedicated when the private gateway is activated.

Private gateways can be dedicated to a single user ID. When a private gateway is mapped to a particular user ID, all connection requests routed through that gateway will be directly sent to that user ID. The connection request is not checked to determine which user ID is the target of the request. This dedicated

private gateway should be defined for private resource server virtual machines that receive requests from multiple users.

A nondedicated private gateway can be used to connect to multiple user IDs. It is not dedicated to a particular user ID. Connection requests routed through a nondedicated gateway are checked to determine the user ID to which the connection request needs to be routed. One nondedicated private gateway could be used by all users who are requesting access to their own virtual machine.

Private gateways may be associated with a Conversation Management Routine (CMR). A CMR is a service pool manager which routes incoming connections from the SNA network to an available service pool virtual machine.

What Is a System Gateway?

A ***system gateway*** is a gateway defined by the VM system (CP) during IPL and provides a way to access resources (global, private, or system) on a specific system within a TSAF or CS collection. Figure 81 on page 476 shows two systems in a TSAF collection both having a system resource X identified. A user on system VM1 can access system resource X on system VM2 through the system gateway of the target system (VM2).

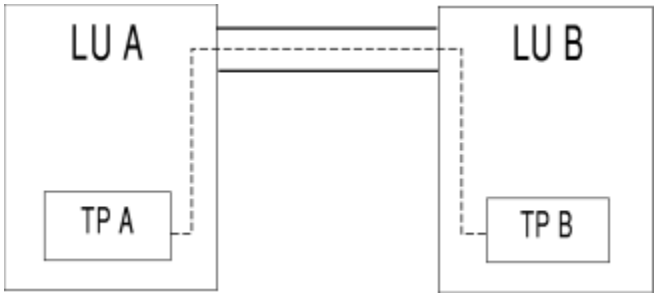


Figure 81. Using a System Gateway to Get System Resources

Similarly, Figure 82 on page 476 shows two systems in a TSAF collection, both with the user ID Z defined, where Z is a private resource manager. User Y on system VM1 can access private resource manager Z on VM2 through the system gateway of the target system (VM2).

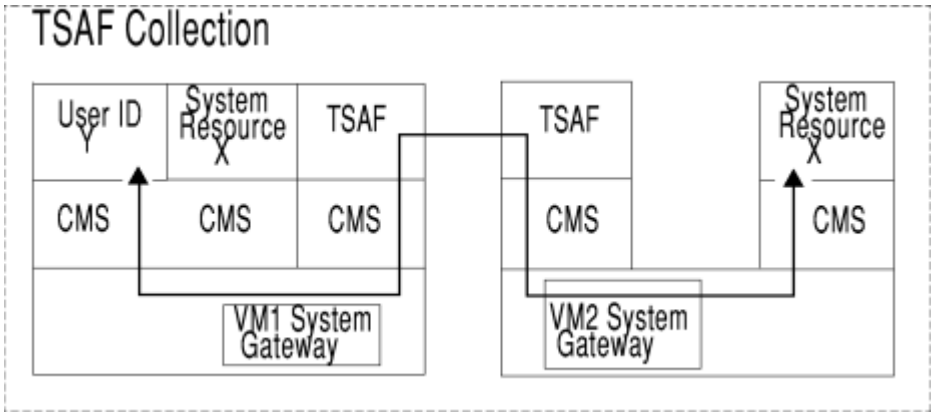


Figure 82. Using a System Gateway to Get to a Private Resource Manager

The system gateway also provides access to private and global resources in a CS collection from an adjacent TSAF collection. Similarly, private and global resources in a TSAF collection can be accessed from an adjacent CS collection, as shown in Figure 83 on page 477.

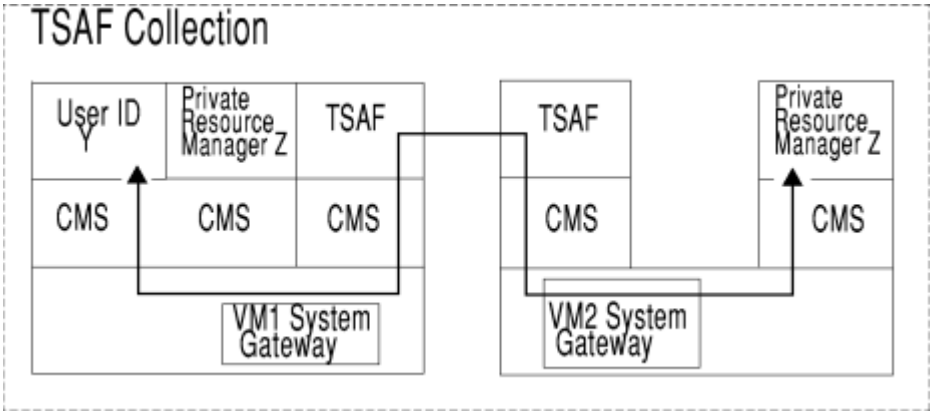


Figure 83. Using a System Gateway to Get Resources on an Adjacent Collection

The system gateway name for a system is identified automatically when CP is initialized on the system. As the default, the system gateway name is the same as the system name that was specified on the SYSTEM_IDENTIFIER configuration statement. This is the recommended naming convention. However, if the default system gateway name conflicts with another gateway name, you can specify a unique gateway name on the SYSTEM_IDENTIFIER configuration statement. See the *z/VM: CP Planning and Administration* for more information about the SYSTEM_IDENTIFIER configuration statement.

Chapter 32. Program-to-Program Communications

This chapter introduces communications programming concepts and presents some overall information you will need to write user programs or resource manager programs.

Basic Concepts

Before we start getting into the details of program-to-program communication, this section provides a foundation for you to understand basic APPC communications concepts.

Communications Partners

In a typical scenario, a user program wants to communicate with a resource manager program. These two programs are called ***communications partners***. In the following figure, A is the partner of B, and B is the partner of A. In a CS collection, user programs can be located on z/VM, DOS, NetWare, Windows®, AIX® , or OS/2® systems. Resource manager programs can be located on z/VM, AIX, OS/2, or NetWare systems.

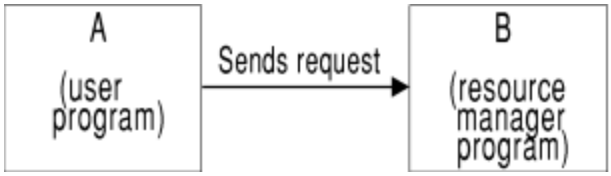


Figure 84. Communications Partners

However, there could be a virtual machine in the middle that allows for communications between the two partner programs. This "middle virtual machine" is called an ***intermediate server***, as shown in the following figure.

Figure 85. Intermediate Servers

For a connection outside your local system within a TSAF collection, the TSAF virtual machine is an intermediate server. For a connection outside your local system within a CS collection, CP is an intermediate server. For a connection outside of the TSAF collection, the AVS virtual machine is an intermediate server. TSAF and AVS are special types of intermediate servers called ***communications servers***. Note that for terminology purposes, even if a request is routed through one or more intermediate servers, A's communications partner is B because the intermediate servers are transparent.

Paths

So that two programs can communicate, a logical connection must be established between them. When this connection is established, the system supplies a name that a program uses to reference the connection. This connection is known as a conversation in APPC. In APPC/VM, this connection between programs is called a ***path***.

States

APPC is a ***half-duplex*** protocol. That means data can be transmitted back and forth between partners, but communications can only go in one direction at a time. For instance, if program A is sending data, its partner, program B, cannot send data until A is done sending.

To let programs communicate, APPC enforces ***conversation states***. These states govern what functions a program can and cannot perform on a conversation.

The basic set of conversation states are:

Reset
The initial state, before communications begin.

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.

Deallocate
The state a program is in when its partner stops communications.

As the program issues functions on a conversation, the state of this conversation can change. The change in the state of the conversation is a result of the function issued, the result of a function issued by a partner or result of a system error. A program with multiple conversations could have each conversation in a different state at any given point in time. For example, one conversation can be in receive state and another in read state concurrently.

The CPI Communications (also known as SAA communications interface) and APPC/VM assembler programming interfaces each implement these basic states in various ways. See Chapter 33, "Understanding CPI Communications," on page 493 for details on CPI Communications. See *z/VM: CMS Application Development Guide for Assembler* for details on using the APPC/VM assembler interface.

Using Basic Communications Functions

Regardless of what programming interface you use, there are three basic steps when communicating with another program:

1. Starting communications
2. Sending and receiving data
3. Ending communications.

You can think of program communications as being analogous to typical, everyday telephone communications. The following sections discuss the three basic steps and draw on the telephone analogy.

Step 1: Starting Communications with Another Program

To start communications, a program must first establish a connection. How your program makes the connection depends on where its target (partner) program is located. When establishing communications, you must identify your partner program, and we will talk more about this later in this chapter. Think of establishing program communications as being analogous to dialing a telephone.

When your program issues a command to connect, your communications partner gets notified. After getting your request to connect, your communications partner can respond in one of two ways:

- Accept the connection, if it wants to communicate with your user program. Think of this as being analogous to answering a ringing telephone, finding out who is calling, then deciding to talk.
- Reject the connection, if it does not want to communicate with your user program. Think of this as being analogous to answering a ringing telephone, finding out it is the wrong number, then deciding to hang up.

The system where the target program resides, and the target program itself, can selectively decide what programs it will communicate with. We will also talk more about this later in this chapter.

As mentioned earlier, your program can connect to resources that are in your same system, your same TSAF collection, your same CS collection, or in an SNA network. The following scenarios show the types of connections for a source program in z/VM.

Requesting to Start Communications with a Program on Your System: If the resource is on your own system (VMSYS1 in the following figure), CP routes you there without the need for ISFC or the TSAF virtual machine. The resource can be either local, global, system, or private.

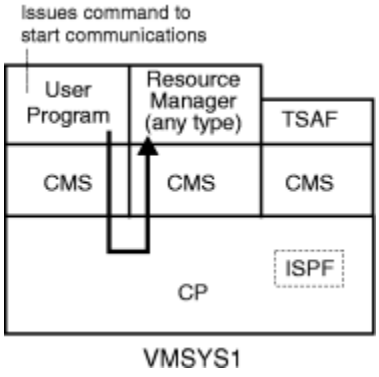


Figure 86. Target Program Located on the Same System

Requesting to Start Communications with a Program on a Different z/VM System: If a virtual machine on another system within the TSAF collection is managing either a global or private resource (VMSYS2 in the following figure), the TSAF virtual machine routes the connection to that resource manager program. Note that a user program on VMSYS1 can connect to a system resource on VMSYS2 through the system gateway of VMSYS2. See *z/VM: Connectivity* for more information.

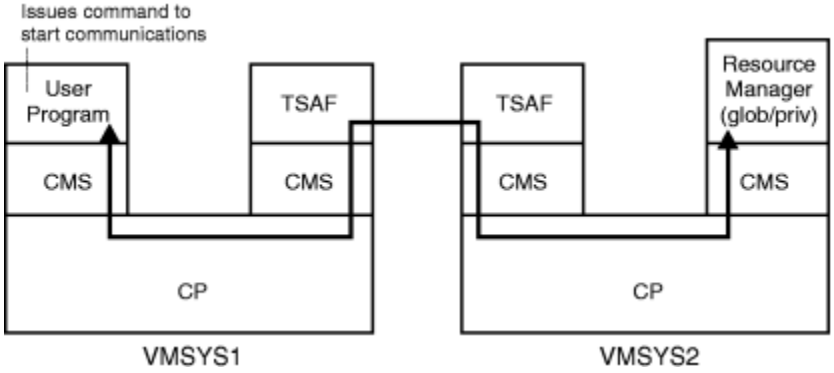


Figure 87. Target Program Located on Another z/VM System

Requesting to Start Communications with a Program in an SNA Network: If your program connects to a resource not in the same TSAF collection, AVS and VTAM route the connection to the appropriate LU in the SNA network. This LU may be another TSAF collection as shown in Figure 88 on page 482.

When trying to start communications with a program outside of your own TSAF collection, you need special allocate data so that the request gets properly routed.

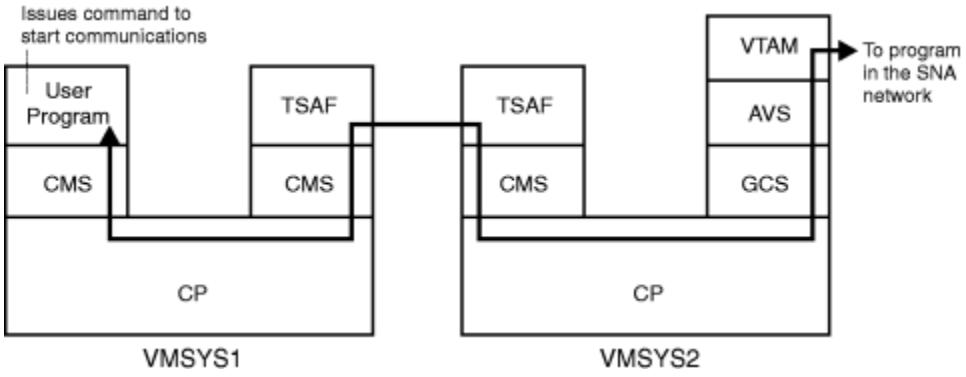


Figure 88. Target Program Located in an SNA Network

Step 2: Sending and Receiving Data

When your program requests to start communications and your communications partner accepts the connection, your program can now send data. As your program sends data, your communications partner is notified, and it can then receive the data. Think of this as being analogous to the talking that takes place in a typical telephone conversation.

Once a connection is established, the requesting program is in Send state for the conversation, and its communications partner is in Receive state. You can only receive data, status, or both when you are in the Receive state. In Send state you can send data or switch to Receive state to allow your partner to send data.

To send data, your program must specify buffers. How you set up the data within those buffers depends on whether your program and your partner are using mapped conversations or basic conversations. Briefly, programs using ***mapped conversations*** just specify data in buffers; programs using ***basic conversations*** must format data into APPC logical records before sending it. Communications partners must agree to which conversation type they will be using before starting communications.

Step 3: Ending Communications with Another Program

When your program is finished communicating with your partner program, your program should end communications with your partner. Your partner will then receive notice that you are finished, and then it will typically finish also. Think of this as being analogous to hanging up the telephone when you are finished talking.

Using Advanced Communications Functions

In addition to starting and ending conversations, and sending and receiving data, your communications programs can take advantage of advanced communications functions of:

- Requesting confirmation
- Signaling an error
- Requesting to send data
- Establishing a protected conversation.

Requesting Confirmation

Before you send more data, switch states or deallocate the conversation, you may request that your communications partner confirm that everything is okay before you continue with the next function.

- When you are sending data to another virtual machine, you can ask your communications partner to confirm that it received the data and you should continue to send data. When your program tries to do this, your partner will do one of the following:
  - Indicate that your program can continue sending data

  - Indicate that something is wrong
  - End communications.
- When you are in Send state and wish to switch to receive state, you can ask your communications partner to confirm that this is okay, before making the state change.

  If you want your program to be able to request confirmation, you must specify a synchronization level of Confirm when you start communications. Think of this as being analogous to a telephone conversation where at specific points in the conversation you will not continue talking until you hear a specific, expected reply from your partner.
- When you are in Send state and wish to end communications, you can ask your communications partner to confirm that this is okay, before the deallocate is done.

Signaling an Error

When you sense that there is an error in the communication, whether you are in Send or Receive state, you can signal your communications partner to cause a break in the normal send/receive sequence.

When sending an error notice to your communications partner, you can also send data that describes the error in more detail to help your partner diagnose the error. Think of this as being analogous to a telephone conversation where you interrupt your partner's speaking, telling him you do not understand what he is talking about.

Requesting to Send Data

At some point, while your partner is sending data, you can let him know that you want to send data. Your partner may choose to ignore your request or agree to your request. If your partner agrees to your request and switches to Receive state, then you would be able to send data. Think of this as being analogous to a telephone conversation where your partner is talking but you interrupt to ask if you can talk.

Establishing a Protected Conversation

When you are updating two or more resources, you should consider using conversations set up with a synchronization level of sync point. Such conversations are called protected conversations and take advantage of CMS's data integrity facility, Coordinated Resource Recovery (CRR). CRR ensures that all participating resources are be updated (if no errors occur) or no updates are made (if an error occurs).

For example, suppose you want to transfer money from an account in one bank to an account in a different bank. CRR would prevent one account decreasing without the other account increasing. That is, if something happened after one account was decreased but before the other account was increased, CRR would notify the partner application to undo changes to the account that was decreased and the two accounts would appear as if nothing occurred.

Identifying Your Communications Partner

For a program initiating a conversation, it must provide certain information to identify its partner. This includes the partner's transaction program name and other information that describes where the partner program is located.

A program can choose to explicitly provide this information when it starts communications. However, a program can also implicitly provide this information by just specifying a ***symbolic destination name***. This symbolic destination name maps to the partner's transaction program name and location information. For VM, this mapping is done by a CMS communications directory file.

Using a CMS Communications Directory

If a communications directory is set up at your installation, your program can simply specify a symbolic destination name to communicate with your partner program. The partner program could be on the same system, in the same TSAF or CS collection, or in a system in an SNA network.

A CMS communications directory is a special NAMES file that maps symbolic destination names to the location and access security information necessary to connect to a target resource. Resources can be relocated without your programs changing, because the location of the resource is transparent to your program—only your communications directory would need to be updated.

There can be two user-configurable levels of communications directory files: a system level and a user level. The system administrator should set up a system-level communications directory file, and it should be in effect when you log on to your virtual machine. However, you can also create your own communications directory file by using XEDIT. You might wish to do this for the following reasons:

- Your programs use symbolic destination names that are not in the system file.
- Your programs need to connect to programs that are not specified in the system file.
- You want to override definitions in the system communications directory. (User-level communications directories are used before system-level ones.)

(*z/VM: Connectivity* contains more details about communications directory files.)

When making your own communications directory file, you must set it up using tags in a NAMES file. You can now create or change your communications directories, SCOMDIR NAMES and UCOMDIR NAMES, using the NAMES command. See the NAMES command usage notes in the *z/VM: CMS Commands and Utilities Reference* for more information. For more information on the structure of NAMES files, see the NAMEFIND command in the *z/VM: CMS Commands and Utilities Reference*.

Table 67 on page 484 summarizes the contents of a CMS communications directory file entry. The first column identifies the tag you enter when you edit the file, and the second column describes the tag's contents.

Table 67. Contents of a CMS Communications Directory File

Tag	Specified Value		
:nick.	Eight-character symbolic destination name for the target resource.		
:luname.	The locally known LU name, which identifies where the resource resides. It contains 2 blank-delimited tokens, each up to 8 characters in length: an LU name qualifier and a target LU name. The values that can be used for each depend on the connection:		
	Connection	**Locally Known LU Name**	**Target LU Name**
	Private resource within the TSAF or CS collection	*USERID	Private resource manager's user ID
	Local or system resource, or to a global resource within the TSAF or CS collection	*IDENT or blank	blank
	Outside the TSAF or CS collection	Defined gateway name	Name of partner's LU
	Global or system resource on a specific system in the CS or TSAF collection	System gateway name of target system	blank
	Private resource on a specific system in the CS or TSAF collection	System gateway name of the target system	Private resource manager's user ID
	Resource at remote LU *rem_luname* using AVS gateway *loc_luname*	*loc_luname*	*rem_luname*

Table 67. Contents of a CMS Communications Directory File (continued)

Tag	Specified Value
:tpn.	The transaction program name as it is known at the target LU. For a local or global resource, this is the resource name identified by the resource manager. For a private resource, this is the nickname specified in the private resource server virtual machine's $SERVER$ NAMES file. For a resource in the SNA network, this is the transaction program name. This resource ID cannot start with a period. The transaction program name '&TSAF' is reserved for TSAF virtual machines that are using APPC links.
:modename.	For connections outside your TSAF or CS collection, this field specifies the mode name for the SNA session connecting the gateway to the target LU. For connections within the TSAF or CS collection, this field specifies a mode name of either VMINT or VMBAT, or it is omitted. Only user programs running in requester virtual machines with OPTION COMSRV specified in their CP directory entry can specify connections with a mode name of VMINT or VMBAT.
:security.	The security type of the conversation (NONE, SAME, or PGM). Security levels are described on page "What Is Conversation Security?" on page 469.
:userid.	The access security user ID, which is required for security type PGM; it is ignored for other security types.
:password.	The access security password, which is required for security type PGM; it is ignored for other security types.
Note: For additional security, you can specify the access security user ID and password on the APPCPASS statement in the source virtual machine's directory, rather than in this file.	

Once you create your own communications directory file, you must put it into effect by using the SET COMDIR command. When a communications directory file is modified, you must use the SET COMDIR command to put the new copy into effect. (See the *z/VM: CMS Commands and Utilities Reference* for details on this command.)

For more information about identifying your communications partner, look at the tables in "Summary of Connections" on page 488.

Resource Manager Programs

When writing a program to manage a resource, you should consider the four types of resources—local, global, system, and private—and their differences. (See "VM Terminology" on page 470 to review the differences between z/VM resources.)

Once you have written a resource manager program, the virtual machine in which it resides must be properly set up and authorized. The system administrator should be the person to do this. (*z/VM: Connectivity* contains the necessary information.)

Local Resource Manager Programs

Local resources are identified only to a particular system in the TSAF or CS collection, and they can only be accessed from within that system. This means that only authorized user programs on that system can access the local resource. You could write a local resource manager program to control access to a device, or data that does not change often and that is easily copied. Examples of local resource managers are programs that control access to:

- A phone directory
- A department printer.

The local resource manager must be authorized to identify a local resource. The local resource manager must be running when a request to access its resource is made. In addition, it should be set up to do its

own security checking based on the incoming user ID when a program tries to establish a connection with security SAME or PGM.

Global Resource Manager Programs

Global resources are identified to both the TSAF and CS collections. They can be accessed by authorized user programs in the TSAF collection, CS collection or in the SNA network. When it is important for all users in your collection to have access to the same resource (because it changes often, for example), the resource could be defined as global. Examples of global resources are:

- DB2 Server for VM databases
- Shared File System (SFS) file pools.

A global resource manager program must be authorized to identify a global resource. The global resource manager must be running when a request to access its resource is made. In addition, it should be set up to do its own security checking based on the incoming user ID when a program tries to establish a connection with security SAME or PGM.

System Resource Manager Programs

System resources are identified only to a particular z/VM system in the TSAF or CS collection, similar to local resources. However, they are accessible from a remote system in an SNA network, provided that AVS is running on the same z/VM system where the system resource resides. A system resource is also remotely accessible from a remote system in a TSAF or CS collection, provided the access is made using the system gateway of the z/VM system where the system resource resides. For more information about AVS or the system gateway, see *z/VM: Connectivity*.

A system resource manager program must be authorized to identify a global resource (global and system resources share the same name space on a z/VM system). The system resource manager must be running when a request to access its resource is made. In addition, it should be set up to do its own security checking based on the incoming user ID when a program tries to establish a connection with security SAME or PGM.

Private Resource Manager Programs

A private resource is known only within its virtual machine. However, it can be accessed by authorized user programs in the TSAF collection, CS collection or in the SNA network. Resources that are not frequently used or that need to be limited to a single user or group of users should be defined as private resources. Examples of private resource managers are programs that control access to

- Plotters
- Departmental files or programs
- CMS minidisk files for a user on a workstation.

A program that issues privileged instructions should also be defined as a private resource manager because the system administrator can control who uses the program and which programs run in a virtual machine that is authorized to issue privileged instructions.

Considerations for Private Resources

For a private server virtual machine, CMS needs to determine if a user is authorized to access the private resource and to determine which private resource manager should be invoked. The $SERVER$ NAMES file is used to accomplish these tasks. The owner of the private server virtual machine registers private resources and explicitly authorizes the user IDs of users allowed to access its private resource in the $SERVER$ NAMES file.

Table 68 on page 487 summarizes the contents of a $SERVER$ NAMES file entry. The first column identifies the tag you enter when you edit the file, and the second column describes the field's usage.

Table 68. Contents of the $SERVER$ NAMES File	
Entry Tag	**Usage**
:nick.	Specifies the eight character name of the private resource. '&TSAF' is the name reserved for TSAF virtual machines that are using APPC links for communication.
:list.	Specifies the user IDs or user ID nicknames of the users (requesters) authorized to use the private resource.
:module.	Specifies the CMS-invokable name of the resource manager (program) of the private resource specified in the nickname field. The file type of the resource manager is either MODULE or EXEC. The private resource name specified as the nickname will be passed to the resource manager as a parameter.

See *z/VM: Connectivity* for more information on the contents of the $SERVER$ NAMES file.

You can now create or change your $SERVER$ NAMES file using the NAMES command. See the NAMES command usage notes in the *z/VM: CMS Commands and Utilities Reference* for more information.

The private server manager program may not need to perform security checking—CMS verifies an incoming user ID for connections attempted with security SAME or PGM. CMS verifies this by checking that the incoming user ID is found in the $SERVER$ NAMES file. If :list. contains "*" the program may need to check security.

When a user program requests a connection to a private resource, the private resource manager program does not have to be running. The private server virtual machine does not even have to be logged on.

If the private server virtual machine is logged on, CMS invokes the private resource manager program when the program is requested. If the private server virtual machine is not logged on, CP will automatically log on (AUTOLOG) the virtual machine, and then CMS invokes the private resource manager program.

In addition, a private resource manager program can be invoked by a name that is different than the requested resource name (TPN). The :module. tag in the $SERVER$ NAMES file specifies such a program name.

Certain conditions must be met for a virtual machine to process requests for private resources in an autologged environment:

- The following CMS commands must be set as follows:
  - SET SERVER must be ON
  - SET FULLSCREEN must be OFF or SUSPEND
  - SET AUTOREAD must be OFF.

  (Note that these commands should be set up in a PROFILE EXEC.)
- Nothing has caused CMS to issue a VM READ or CP READ.
- The console stack is empty.
- CMS is not in CMS subset mode.
- CMS is at the `Ready ; ` status.

If interactive work needs to be done on a private server virtual machine, the SET SERVER OFF command should be issued so that any connection requests for private resources are rejected.

See “Scenario 2: Request for a Private Resource” on page 509 for an example of a private resource manager scenario using CPI Communications routines. See *z/VM: CMS Application Development Guide for Assembler* for an example of a private resource manager using the assembler APPC/VM interface. The *z/VM: Connectivity* contains detailed information about setting up private server virtual machines.

Intermediate Servers

You can also write a program that does not actually manage a resource, but that controls access to another program that does manage a resource. This type of *middle* program resides in an intermediate server virtual machine.

As with server virtual machines that manage resources, intermediate server virtual machines must be properly set up and authorized. The system administrator should be the person to do this. (*z/VM: Connectivity* contains the necessary information.)

An intermediate server can set up to intercept all connection requests that are intended for a final target —a resource manager program. When the program in an intermediate server gets such a connection request, it must then make its own connection to the final target (the resource manager program). When the intermediate server makes this connection on behalf of a source program, it should forward the user ID of the requesting virtual machine as the access security user ID—not the user ID of its own virtual machine. The intermediate server can specify the original source user ID in CPI Communications and in the APPC/VM assembler interface.

In addition, an intermediate server should also validate incoming data that a source program sends. This is because a source program could have the proper authorization to connect to the intermediate server, but could accidentally or maliciously send incorrect data.

Writing Versatile Programs

You can write an APPC/VM or CPI Communications program to be independent of its partner's location. A program can basically be written the same way, regardless of whether its partner is on the same z/VM system, different z/VM system, or other system in an SNA network.

However, a program that works when communicating with a program on the same system may not work the same way when communicating with a program on a different system. To ensure that an application running within a system continues to run when communicating outside of a single system, keep these guidelines in mind:

- Write your application to handle all possible completion indications and return codes for each verb (APPC function) your application issues. Do not assume that only a subset of completion indications can occur. Return codes are documented with each verb, whether it be a CPI Communications routine or an APPC/VM assembler function.
- In general, you cannot determine whether completion of a function means that your partner has processed or even received your program's request.

  For example, your program should not assume that completion of its APPCVM CONNECT function or CPI Communications Allocate (CMALLC) routine means that your partner has accepted the request.
- When intermediate communications servers like TSAF and AVS handle communications, a program cannot determine when APPC/VM data and indications get to its partner program. The only way to ensure that you can have complete synchronization with your partner program is to use functions that request confirmation from your partner. For example, using the APPC/VM functions SENDCNF (Send Confirm) and SENDCNFD (Send Confirmed) or the CPI Communications routines Confirm (CMCFM) and Confirmed (CMCFMD).

To read more about the general APPC program interface, see the *Systems Network Architecture Transaction Programmer's Reference Manual for LU Type 6.2* book.

Summary of Connections

Table 69 on page 489 summarizes how a locally known LU name and resource name are specified in VM. If a CMS communications directory is used, the user program can specify a symbolic destination name to identify the locally known LU name and target resource name. If a CMS communications directory is not used, the user program must identify the complete locally known LU name and target resource name. For a description of a CMS communications directory and information on how to set one up, see the *z/VM: Connectivity*.

Table 69. Identifying the Target of a Connection Request

Type of Connection		Locally Known LU Name		TPN	Name Space	LU
From	**To**	**LU Qualifier**	**Target LU Name**			
A program in a CS collection or a virtual machine in the TSAF collection.	The local resource or system resource on the local system, if they exist, or the global resource in the collection, if it exists.	*IDENT	0	Local, Global, or System resource name	Local, Global, or System	TSAF or CS collection
A program in a CS collection or virtual machine in the TSAF collection.	The *userid* on the local system, if it exists, or the *userid* in the collection, if it exists. If more than one *userid* exists in the collection, the one that is logged on (if one is) receives the connection request; otherwise the first one found in the collection will be chosen. The requester must be authorized in the *userid*s $SERVER$ NAMES file.	*USERID	*userid*	Private resource name	Private	Virtual machine *userid*
A program in a CS collection or virtual machine in the TSAF collection.	The global or system resource on the system identified by *sysgate* in the collection, if it exists, or a global resource in an adjacent CS or TSAF collection, respectively, if it exists and *sysgate* is the system gateway of the node that is common to both collections.	*sysgate*	0	Global or System resource name	Global, System	VM system identified by *sysgate*

Table 69. Identifying the Target of a Connection Request (continued)

Type of Connection		Locally Known LU Name		TPN	Name Space	LU
From	To	LU Qualifier	Target LU Name			
A program in a CS collection or virtual machine in the TSAF collection.	The *userid* on the system identified by *sysgate* in the collection or the *userid* in an adjacent CS or TSAF collection, if it exists, and the *sysgate* is the system gateway of the node that is common to both collections. The requester must be authorized in the $SERVER$ NAMES file of the *userid*.	*sysgate*	*userid*	Private resource name	Private	Virtual machine *userid*
A program in a CS or TSAF collection connected to the target collection through VTAM and AVS.	The system resource on the same VM system as AVS is running in the TSAF or CS collection if it exists, otherwise the global resource in the TSAF or CS collection if it exists.	*loc_gat* AVS global gateway	*rem_gat* AVS global gateway	Global or System resource name	Global, System	TSAF or CS collection
The user ID tied to *loc_gat* in a CS or TSAF collection connected to the target collection through VTAM and AVS.	The user ID tied to *rem_gat* on the same VM system as AVS is running, if it exists, or the user ID in the TSAF or CS collection if it exists. The same search criteria applies here as for *USERID *userid*.	*loc_gat* AVS private dedicated gateway	*rem_gat* AVS private dedicated gateway	Private resource name	Private	Virtual machine user ID tied to *rem_gat*.

Table 69. Identifying the Target of a Connection Request (continued)

Type of Connection		Locally Known LU Name		TPN	Name Space	LU
From	To	LU Qualifier	Target LU Name			
A program in a CS or TSAF collection connected to the target collection through VTAM and AVS.	The access user ID on the same VM system as AVS is running in the TSAF or CS collection, if it exists, or the access user ID in the TSAF or CS collection, if it exists. The same search criteria applies here as for *USERID *userid*.	*loc_gat* AVS private nondedicated gateway	*rem_gat* AVS private nondedicated gateway	Private resource name	Private	Virtual machine of the access user ID.
The user ID tied to *loc_gat* in a CS or TSAF collection connected to the target collection through VTAM and AVS.	The access user ID on the same VM system as AVS is running in the TSAF or CS collection, if it exists, or the access user ID in the TSAF or CS collection, if it exists. The same search criteria applies here as for *USERID *userid*.	*loc_gat* AVS private dedicated gateway	*rem_gat* AVS private nondedicated gateway	Private resource name	Private	Virtual machine of the access user ID.
A program in a CS or TSAF collection connected to the target collection through VTAM and AVS.	The user ID tied to *rem_gat* on the same VM system as AVS is running, if it exists or the user ID the TSAF or CS collection, if it exists. The same search criteria applies here as for *USERID *userid*.	*loc_gat* AVS private nondedicated gateway	*rem_gat* AVS private dedicated gateway	Private resource name	Private	Virtual machine user ID tied to *rem_gat*.

Which Programming Interface Do You Want to Use?

If you intend to write your communications programs using CPI Communications, continue with Chapter 33, "Understanding CPI Communications," on page 493. If you intend to write your communications programs using the APPC/VM assembler interface, refer to the *z/VM: CMS Application Development Guide for Assembler*.

Chapter 33. Understanding CPI Communications

Common Programming Interface (CPI) Communications (also known as SAA communications interface) is an SAA defined interface you can use to write APPC communications programs in REXX, assembler or high-level programming languages. CPI Communications defines a set of routines that programs can use so that they will be more portable to other systems that abide by SAA's definitions.

In addition, z/VM has defined some routines that are extensions to CPI Communications. These z/VM extension routines are useful to exploit the capabilities of the z/VM operating system. In this book, the term "CPI Communications" includes the SAA routines and z/VM's extension routines.

Like the APPC/VM assembler interface, CPI Communications lets your program communicate with another program that is on the same z/VM system, on a different z/VM system, or in a network defined by SNA. Both communicating programs (called partner programs) do not have to use CPI Communications routines. However, writing programs using CPI Communications routines is easier than coding programs with APPC/VM assembler functions, for two important reasons:

1. CPI Communications routines do not have to be called from assembler language programs. Programs written in REXX or high-level languages such as COBOL can take advantage of this communications interface.
2. The CPI Communications definition generally includes fewer parameters on routines.

Throughout this chapter, assume that both partner programs are using CPI Communications routines.

Basics of CPI Communications

CPI Communications is based on the APPC half-duplex protocol, so it enforces states. At a given point in time, each conversation is always in a particular state, and a program can only issue certain functions from that state. The CPI Communications conversation states, which are an expansion of the APPC states defined earlier in this book, are as follows:

- Reset
- Initialize
- Send
- Receive
- Send-Pending
- Confirm
- Confirm-Send
- Confirm-Deallocate
- Defer-Receive
- Defer-Deallocate
- Sync-Point
- Sync-Point-Send
- Sync-Point-Deallocate.

APPC sees a logical connection between programs as a conversation. Each program refers to the conversation by a system-defined value. In CPI Communications, this value is called a conversation ID.

CPI Communications defines default values for many parameters that determine how routines execute. Your program can override these default values by calling various Set routines. In addition, your program can also examine the current value of many of these parameters by calling various Extract routines.

CPI Communications also defines pseudonym character strings for constants that make your programs easier to read. These character strings are prefixed with "CM" or "XC", and are equated to integer

values. For instance, you can use the character string CM_PARAMETER_ERROR to check for a return code of 19. See the VM appendix in the *Common Programming Interface Communications Reference (https://publibfp.dhe.ibm.com/epubs/pdf/c2643999.pdf)* for information on the pseudonym files provided by VM.

Invoking CPI Communications Routines

In z/VM, CPI Communications can only be used in a CMS environment. The following SAA languages can be used on z/VM to call CPI Communications routines and the VM extension routines:

- Application generator (Cross System Product implementation)
- C
- COBOL
- FORTRAN
- PL/I
- REXX (SAA).

In addition, the following non-SAA languages can be used on z/VM:

- Assembler
- Pascal.

See *Common Programming Interface Communications Reference (https://publibfp.dhe.ibm.com/epubs/pdf/c2643999.pdf)* for complete details on coding CPI Communications routines and the z/VM routines that are extensions to the CPI Communications routines.

Invocation Errors

If CMS cannot execute the CPI Communications routine your program is trying to call, the following error message is generated:

```
1292E  Error calling CPI-Communications routine, return code retcode
```

See the *z/VM: CMS and REXX/VM Messages and Codes* for details on return codes for message DMS1292E.

Using Basic CPI Communications Functions

To begin writing CPI Communications programs, you need to know the CPI Communications 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 first call the Initialize_Conversation (CMINIT) routine with a name that identifies the target program. This name can be one of the following:

- A symbolic destination name—This is used as an index to a CMS communications directory file that contains information necessary to establish the connection. Note that the CMS communications directory file is called "side information" by SAA.
- The actual target transaction program name or resource ID—Note that the CMS communications directory file is still searched for the specified name. Therefore, this name is used as the transaction program name only if it cannot be resolved using the CMS communications directory.

The Initialize_Conversation routine sets various conversation characteristics from the CMS communications directory file (if available) and sets other conversation characteristics to default values.

After initializing a conversation, your program must call the Allocate (CMALLC) routine to request a connection. Before calling Allocate, however, a program can change the conversation characteristics that were set by Initialize_Conversation by calling Set routines. Some of the Set routines allow you to override characteristics that could have been initialized from the CMS communications directory:

- Mode name
- TP name
- Partner LU name
- Conversation security type
- Access security user ID
- Access security password.

Before calling the Allocate (CMALLC) routine, your program must decide how data will be sent on the conversation (specified by *conversation_type*) and whether synchronization routines will be used in the conversation (specified by *sync_level*). You can use the Set functions to change the *conversation_type* characteristic and the *sync_level* characteristic. Your partner program, after it Accepts (CMACCP) the conversation, can then extract the conversation_type and sync_level characteristics to decide whether it can handle your choice for these characteristics.

The default conversation type is mapped conversations. Mapped conversations allow you to send and receive user data without worrying about APPC logical record formats. However, you can change the conversation type to be a basic conversation. Basic conversations require programs to exchange data in a standardized format.

The default *sync_level* value is CM_NONE. CM_NONE specifies that the program will not use synchronization routines for the conversation. The *sync_level* characteristic can be changed to CM_CONFIRM or CM_SYNC_POINT. CM_CONFIRM specifies that the program will perform confirmation processing on the conversation specified. CM_SYNC_POINT specifies that the conversation is to be a protected conversation. This means that Coordinated Resource Recovery (CRR) will coordinate commits (and backouts) of work among multiple protected resources in a distributed application.

For more information on starting a conversation, see *Common Programming Interface Communications Reference (https://publibfp.dhe.ibm.com/epubs/pdf/c2643999.pdf)*.

Sending and Receiving Data on the Conversation

After you call the Allocate (CMALLC) routine:

- The conversation is in **Send** state for your program
- The conversation type and sync level have been established.

You can now send data, calling the Send_Data (CMSEND) routine. You set up buffers to send data to your partner program according to the conversation type. Also, before sending data, a program can set the *send_type* characteristic for the conversation.

When your communications partner program is started, it accepts the conversation by calling Accept_Conversation (CMACCP). After calling the Accept_Conversation routine, the conversation is in **Receive** state for your partner's program. Your partner receives the data by calling the Receive (CMRCV) routine. Before receiving data, a program can set the *receive_type* and (for a basic conversation) the *fill* characteristics for the conversation.

Ending a Conversation

When your program is finished communicating with your partner program, you should end the conversation by calling the Deallocate (CMDEAL) routine. Before ending the conversation, a program can set the *deallocate_type* characteristic for the conversation. When your partner program receives a deallocation notice, the program cannot perform any further functions on that conversation.

Using Advanced CPI Communications Functions

In addition to the basic functions, CPI Communications 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 use Coordinated Resource Recovery (CRR) support.

Requesting Confirmation

When you are sending data to your partner program, you may want to synchronize execution with your partner and request confirmation that you should continue to send. To do this, you call the Confirm (CMCFM) routine. The Confirm is not complete until your partner calls one of the following routines:

- Confirmed (CMCFMD) to indicate that the sender can continue
- Send_Error (CMSERR) to indicate that something is wrong
- Deallocate (CMDEAL) to end communications.

To call Confirm and Confirmed, the program starting the conversation must indicate that confirmation is allowed on the conversation. The program does this by calling a Set_Sync_Level (CMSSL) to set the *sync_level* characteristic for the conversation to CM_CONFIRM or CM_SYNC_POINT. This call must be made after calling Initialize_Conversation and before calling Allocate.

Signaling an Error

If your program determines that there is an error in the communications, whether the conversation is in **Send** or **Receive** state, your program can call the Send_Error (CMSERR) routine. This signals your communications partner and causes a break in the normal send/receive sequence.

If you are in **Receive** state and issue Send_Error:

1. The error notice goes to your communications partner.
2. Your virtual machine enters **Send** state.

If you are in **Send** state and issue Send_Error:

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 if the conversation type is basic. Your log data should provide information that describes the error in detail to help your partner diagnose the error. To specify log data, your program must call the Set_Log_Data (CMSLD) routine before calling Send_Error. If your partner is not using CPI Communications, however your partner must have indicated that it will accept log data. (Note that you can also send log data when calling the Deallocate (CMDEAL) routine.)

Requesting to Send Data

At some point, when your program is receiving data, you may want to inform your partner that you want to send data. To do this, you can call Request_To_Send (CMRTS). However, note that your partner can choose to ignore your Request_To_Send.

Your program will be able to send data when you are informed that your partner has changed states. Your partner can change states by doing one of the following:

- Calling the Receive (CMRCV) routine
- Calling Set_Send_Type (CMSST) to set the *send_type* characteristic to CM_SEND_AND_PREP_TO_RECEIVE, and then calling the Send_Data (CMSEND) routine
- Calling the Prepare_To_Receive (CMPTR) routine.

Establishing a Protected Conversation

A protected conversation allows your distributed application to take advantage of the z/VM data integrity facility, Coordinated Resource Recovery (CRR). CRR provides services to coordinate updates to two or more resources. See Chapter 16, “Your Applications and Data Integrity,” on page 241 for more details. Protected conversations extend this protection to resources on remote systems so that the commit or rollback of all changes for all conversation partners in a logical unit of work can be coordinated. Setting a conversation's *sync_level* characteristic to CM_SYNC_POINT before allocation indicates to

CPI Communications that you want a protected conversation. See the *Common Programming Interface Communications Reference (https://publibfp.dhe.ibm.com/epubs/pdf/c2643999.pdf)* for more details on setting up and using protected conversations.

Using VM Extensions to CPI Communications

z/VM provides some routines that are extensions to SAA CPI Communications. Programs using these routines will require modification to be portable to other SAA systems. However, these routines can be used to take advantage of the z/VM operating system. These extension routines are briefly introduced in the sections that follow.

Security

The default security value for CPI Communications conversations is SAME (XC_SECURITY_SAME). In z/VM, two additional security levels, NONE (XC_SECURITY_NONE) and PGM (XC_SECURITY_PROGRAM) are available. z/VM provides a routine called Set_Conversation_Security_Type (XCSCST) that lets a program explicitly specify the security value for the conversation. The security type also can be set in side information by using the `:security.` tag.

When the security type is PGM, an access security user ID and password must be provided. They can be supplied on the `:userid.` and `:password.` tags in side information if `:security.`PGM has been specified there. These values can also be provided explicitly within a program by calling the z/VM-provided routines Set_Conversation_Security_User_ID (XCSCSU) and Set_Conversation_Security_Password (XCSCSP). The access security user ID associated with a conversation can be obtained with the Extract_Conversation_Security_User_ID (XCECSU) routine.

If there are concerns about placing security information in a file, the values can be provided in an APPCPASS statement in the virtual machine's CP directory. Entries in an APPCPASS statement do not override a value provided in either side information or on an explicit Set call.

Values provided on an explicit security Set call override any corresponding information in side information and take precedence over information in an APPCPASS statement for the conversation. However, when the security type is PGM, if only a user ID is provided or neither a user ID nor a password is provided either in side information or with an explicit Set call, then the CP directory is checked for an APPCPASS statement to supply the missing information. A CM_PRODUCT_SPECIFIC_ERROR occurs if the security type is PGM and only a security password is provided.

See *z/VM: Connectivity* for further information on the security types provided by z/VM.

Resource Manager Programs

z/VM provides routines that allow a resource manager application to manage one or more resources and accept more than one conversation per resource. These routines are:

- Identify_Resource_Manager (XCIDRM), which lets an application define the name of a resource it wishes to manage.
- Terminate_Resource_Manager (XCTRRM), which lets an application end management of a resource it had previously defined with Identify_Resource_Manager.

The resource manager application must indicate to CPI Communications its intent to manage a resource by calling the z/VM extension routine Identify_Resource_Manager. Having done this, the resource manager application can call the Wait_on_Event (XCWOE) routine to wait for allocation requests to the resource being managed. When ending the resource manager application, the Terminate_Resource_Manager routine should be called. Failure to call Terminate_Resource_Manager will result in the name of the resource remaining active until CMS end-of-command processing.

The resource manager application can be a program that either has been started automatically as a result of an allocation request or has been started by local (operator) action. When started automatically as a result of an allocation request, if the resource manager application wishes to declare its intent to manage a resource, it must call Identify_Resource_Manager **before** either accepting the

conversation that started it (using Accept_Conversation) or initializing any conversation characteristics (using Initialize_Conversation). Otherwise, an Identify_Resource_Manager call will not be allowed.

Global, Local and System

For global, local, and system resource managers, the application must be started and the resources identified (using Identify_Resource_Manager) before another application can attempt to allocate a conversation for those resources successfully.

Private

A private resource manager virtual machine can be autologged and the private resource manager application automatically invoked as a result of a private resource connection request. When CMS is in the `Ready;` state for the private resource program, CMS invokes the specified private resource program. The resource name is passed to the application as a parameter. For example, a REXX private resource manager program can be coded like this:

```
/* This is an example of a private resource manager application.   */

arg resource_name     /* private resource name passed as parameter */

/* Equate pseudonyms to integer values based on CMREXX COPY file.  */

address command 'EXECIO * DISKR CMREXX COPY * (FINIS STEM PSEUDONYM.'
do index = 1 to pseudonym.0
  interpret pseudonym.index
end

resource_manager_type = xc_private  /* want a private resource     */
service_mode = xc_multiple          /* handle more than 1 at a time*/
security_level_flag = xc_reject_security_none

address cpicomm 'XCIDRM resource_name resource_manager_type',
                'service_mode security_level_flag return_code'

if rc = 0           /* any csl errors? */
then do
  if return_code = cm_ok
  then do           /* don't have to wait for first connection */
                    /* go accept the conversation              */
    address cpicomm 'CMACCP conversation_id return_code'
     .
     .
     .
```

Note: Following a successful Identify_Resource_Manager call, the application could have called Wait_on_Event (XCWOE) before calling Accept_Conversation; the Wait_on_Event would complete immediately with *event_type* set to XC_ALLOCATION_REQUEST.

For a discussion of z/VM resources and resource manager programs see *z/VM: Connectivity*.

Considerations for TP-Model Applications in z/VM

SAA CPI Communications provides a programming interface to IBM's SNA LU 6.2. The set of calls defined by SAA, however, does not implement every aspect of the LU 6.2 protocol. z/VM provides extensions to SAA CPI Communications to support several additional LU 6.2 features, such as support for security types. z/VM also provides routines that are considered extensions to the LU 6.2 architecture. Resource manager support for accepting multiple incoming conversations, for example, is not part of the LU 6.2 protocol.

This section describes how CPI Communications applications in the z/VM environment can establish conversations that closely conform to the LU 6.2 model for communications. Such applications are referred to here as LU 6.2 transaction program model applications, or *TP-model applications*. While a TP-model application can be created using only SAA CPI Communications routines, such an application is also allowed to call most of the z/VM extension routines.

LU 6.2 Communications Model

In LU 6.2, LUs initiate and run transaction programs. A transaction program (TP) initiates a conversation with its TP partner using the services of the LUs. In Figure 89 on page 499, TP A in LU x allocates a conversation to TP B in LU y. LU x formats and presents an allocation request in the form of an LU 6.2 Attach to LU y. LU y validates the Attach and starts a new instance of TP B.

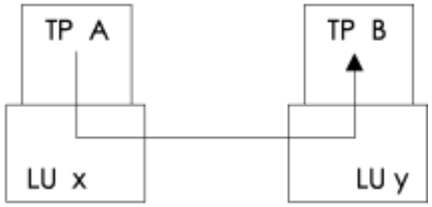


Figure 89. LU 6.2 Communications Model

In this example, both TP A and TP B are TP-model applications:

- TP A is the initial program of a distributed application. It is invoked by some process other than an Attach, typically by an end-user command. TP A has no incoming conversations.
- TP B is invoked as a result of the Attach presented to LU y. There is one and only one incoming conversation.

TP A and TP B can allocate any number of conversations.

z/VM TP-Model Applications

In Figure 90 on page 499, assume that the allocation of a conversation by program A in virtual machine VMUSR1 causes program B to be automatically started by CMS in virtual machine VMUSR2.

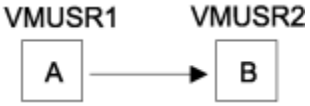


Figure 90. Creating a TP-Model Application in z/VM

Both programs basically determine whether they are TP-model applications by the CPI Communications calls that they issue. Program A is considered a TP-model application so long as it does not issue the Identify_Resource_Manager (XCIDRM) call. Program B's classification, though, is determined by how it is started and by the first successful CPI Communications call that it issues. Calling Accept_Conversation (CMACCP) or Initialize_Conversation (CMINIT) will result in program B being a TP-model application. If it is not desirable for program B to be a TP-model application, then it should issue Identify_Resource_Manager as its first CPI Communications call.

z/VM considers an application like program A in Figure 90 on page 499 to be a TP-model application if it has the following characteristics:

- The application is started by CMS as a result of local (operator) action.
- There are no incoming conversations. However, the application can allocate any number of conversations.
- The Identify_Resource_Manager z/VM extension routine has not been called.

z/VM considers an application like program B in Figure 90 on page 499 to be a TP-model application if it has the following characteristics:

- The application is started automatically by CMS as a result of a private resource connection request.
- There is only one incoming conversation, which is the conversation that caused CMS to start the application. However, the application can allocate any number of conversations.
- The first successful CPI Communications call is to either the Accept_Conversation routine or the Initialize_Conversation routine.
- The Identify_Resource_Manager extension routine has not been called.

Implications

Certain behavior is enforced and certain services are provided by z/VM for TP-model applications:

- For TP-model applications like program B in Figure 90 on page 499, there are three z/VM extension routines that the application is not allowed to call. These routines are: Identify_Resource_Manager, Terminate_Resource_Manager (XCTRRM), and Set_Client_Security_User_ID (XCSCUI). A call to one of these three routines results in a *return_code* of CM_PROGRAM_STATE_CHECK.
- The virtual machine in which a TP-model application like program B in Figure 90 on page 499 is running may require authorization to issue DIAGNOSE code X'D4'. This authorization is necessary if the TP-model application is an intermediate server allocating a conversation with a *conversation_security_type* characteristic of XC_SECURITY_SAME, which is the default value (and the only security type supported by SAA CPI Communications). In that case, the access security user ID provided by the incoming conversation that caused CMS to start the application is automatically propagated on the allocated conversation (assuming the application performs no CMS work unit manipulation).

  See the section entitled "Considerations for Intermediate Servers" on page 500 for more information on the propagation of access security user IDs.
- For CPI Communications protected conversations (those with the *sync_level* characteristic set to CM_SYNC_POINT), screening is performed for TP-model applications to prevent allocation wrapback. Allocation wrapback occurs when an application tries to allocate a protected conversation whose logical unit of work identifier (LUWID) is already associated with another protected conversation to which the target program is a partner. Such a situation can result in deadlock during sync-point processing.

  For example, assume that protected conversations between program A and program B and between program B and program C have been established, as illustrated by the solid lines in Figure 91 on page 500. Three cases of allocation wrapback are illustrated by the dotted lines in that figure. In all of these cases, allocations are being attempted for protected conversations with the same LUWID. Note that the work unit in effect when Allocate (CMALLC) is called determines the LUWID for a protected conversation. See "Notes for Distributed Application Programs" on page 248 for a description of the relationship between LUWIDs and CMS work units.

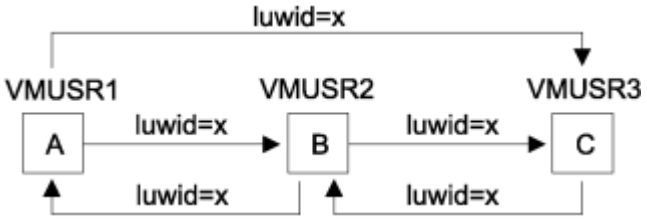


Figure 91. Three Potential Conversation Wrap-Back Scenarios

An allocation request that would result in allocation wrapback is automatically rejected for TP-model applications by CMS in the target virtual machine. If an application allocates a protected conversation that would result in allocation wrapback, it receives a *return_code* value of CM_TP_NOT_AVAILABLE_NO_RETRY on a subsequent call.

Considerations for Intermediate Servers

An intermediate server is a program that handles communications requests to a resource manager program on behalf of a user program. For example, if program A allocates a conversation to program B and program B in turn allocates a conversation to program C on behalf of program A (A→B→C), program B is considered an intermediate server.

An intermediate server, such as program B, allocates the conversation to the remote program, program C, with a *conversation_security_type* of xc_security_same. The access security user ID flowed to program C depends on whether the intermediate server is considered by z/VM to be a TP-model application (as described in the section entitled "Considerations for TP-Model Applications in z/VM" on page 498).

Note that if the *conversation_security_type* characteristic for the conversation between program A and program B is set to XC_SECURITY_NONE and that for the conversation between program B and program C is XC_SECURITY_SAME, the security type for the latter conversation is effectively XC_SECURITY_NONE.

TP-Model Application

If the intermediate server is considered to be a TP-model application and the conversation between program A and program B has a *conversation_security_type* of XC_SECURITY_SAME, then z/VM flows the user ID of the virtual machine in which program A is running (VMUSR1) to the target partner, program C, as shown in Figure 92 on page 501.

Figure 92. Access Security User ID of User Program Flowed from VMUSR1 to VMUSR3

If the conversation allocated between program A and program B has a *conversation_security_type* of XC_SECURITY_PROGRAM, then the access security user ID that flows to program C is the user ID set by program A with a call to Set_Conversation_Security_User_ID (XCSCSU) or provided by program A's virtual machine either in side information or in an APPCPASS statement.

To propagate the access security user ID associated with the inbound conversation, the virtual machine running a TP-model intermediate server must be authorized to issue DIAGNOSE code X'D4'. If the virtual machine does not have the appropriate authorization, an attempt by a TP-model intermediate server to allocate a conversation with a *conversation_security_type* of XC_SECURITY_SAME will fail with a *return_code* of CM_PRODUCT_SPECIFIC_ERROR.

There are two cases, however, when the intermediate server virtual machine does not need to issue the DIAGNOSE code to propagate an access security user ID and thus does not need authorization:

- When the inbound access security user ID matches the logon user ID for the TP-model intermediate server
- When the inbound connection to the intermediate server is made with an alternate user ID that matches the access security user ID. In this case, the alternate user ID is assigned to the intermediate server's virtual machine upon accepting the conversation. If the alternate user ID does not match the access security user ID, the program will not be able to allocate a conversation as an intermediate server. See *z/VM: Connectivity* for a description of connections to service pool virtual machines with an alternate user ID.

Note that a TP-model application typically does not manipulate CMS work units. If the intermediate server has obtained any additional CMS work units, however, it must ensure that the work unit associated with the conversation with program A is the current default work unit when allocating the conversation to program C. If, when Allocate is called, the current default work unit is different from the work unit associated with the conversation and the *conversation_security_type* is XC_SECURITY_SAME, program B's user ID will be flowed to program C, as shown in Figure 94 on page 502.

TP-Model Intermediate Servers and Assigned Alternate User IDs

If a private resource connection request with an alternate user ID starts a TP-model intermediate server application (for example, a service pool application), then the application must accept the connection before allocating any outbound conversations.

If the access security user ID presented to the TP-model intermediate server application does not match the alternate user ID assigned to the virtual machine, then any attempts to issue Allocate (CMALLC) to establish an outbound conversation will fail with a *return_code* of CM_PRODUCT_SPECIFIC_ERROR until the inbound connection that assigned the alternate user ID is terminated.

Non-TP-Model Application

If the intermediate server is not a TP-model application, the access security user ID flowed to program C when *conversation_security_type* is XC_SECURITY_SAME is the user ID of the intermediate server (VMUSR2), as shown in Figure 94 on page 502.

Figure 93. Access Security User ID of User Program Flowed from VMUSR1 to VMUSR3

Non-TP-model applications often handle multiple inbound conversations, each of which is associated with a different work unit. For program B to act as an intermediate server (as in Figure 92 on page 501) in this scenario, it must obtain the access security user ID of the conversation with program A. It then uses this access security user ID to set the client security user ID that will be flowed to program C.

To obtain the access security user ID for the conversation with program A, program B must issue the Extract_Conversation_Security_User_ID (XCECSU) call. To flow program A's access security user ID, VMUSR1 in Figure 94 on page 502, the intermediate server then must call Set_Client_Security_User_ID (XCSCUI). In addition, the intermediate server must ensure that the work unit associated with the conversation with program A is the current default work unit when Allocate is called. Note that the virtual machine in which program B is running (VMUSR2) must have authorization to issue DIAGNOSE code X'D4' if Set_Client_Security_User_ID is called.

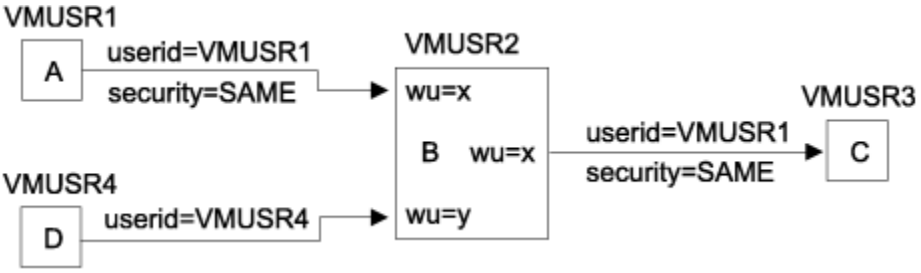


Figure 94. Access Security User ID of Intermediate Server (VMUSR2) Flowed to VMUSR3

CMS Work Units

Because all CPI Communications conversations are associated with CMS work units in z/VM, an extension routine, Extract_Conversation_Workunit_ID (XCECWU), is provided to allow applications to obtain the CMS work unit ID for a given conversation. After issuing an Allocate or Accept_Conversation call to establish a conversation, Extract_Conversation_Workunit_ID is used to extract the CMS work unit ID. The CMS work unit ID can be specified on such CSL routines as DMSPUSWU (Push Default Work Unit ID), DMSCOMM (Commit), and DMSROLLB (Rollback).

Note that there is no provision for work unit manipulation in SAA; programs that need to be portable to other SAA operating environments should use the default work unit. Using the default work unit simply means never specifying a work unit ID on CSL routines with the *workunitid* optional parameter and never manipulating (getting, pushing, or popping) work unit IDs in the applications. See “Using Work Units in Application Programs” on page 133 for more information on CMS work units.

z/VM Resource Recovery

CMS provides a data integrity facility called Coordinated Resource Recovery (CRR) to coordinate work among multiple protected resources. Distributed applications can take advantage of this support by using protected conversations (*sync_level* set to cm_sync_point).

To participate in CRR, a resource must be able to register an adapter with the CRR recovery server. Four routines provide functions for programming to the resource recovery adapter interface:

- Extract_Conversation_Security_User_ID (XCECSU)
- Extract_Local_Fully_Qualified_LU_Name (XCELFQ)

- Extract_Remote_Fully_Qualified_LU_Name (XCERFQ)
- Extract_TP_Name (XCETPN).

See Chapter 18, “Getting a Resource Manager to Participate in CRR,” on page 255 for information about programming to the CRR interface.

In addition, the Extract_Conversation_LUWID (XCECL) routine can be used to identify the most recent sync point.

Managing CPI Communications Events in a Virtual Machine

Your program can call the Wait_on_Event (XCWOE) routine to wait for communications from one or more partners. This routine allows your program to wait for and then take action according to the type of request it receives.

If your program needs to know about certain events that occur in its virtual machine, you can provide an interrupt handler that gets control when those events occur. The interrupt handler can then call Signal_User_Event (XCSUE) to queue the event, called a user event. This user event will be reported when your communications program issues Wait_on_Event (XCWOE).

See “Scenario 4: Signaling a User Event” on page 515 for an example of how an application can use the Signal_User_Event routine. See the *z/VM: CMS Application Development Guide for Assembler* for information on writing interrupt handlers.

For multitasking applications, directly handle the VMCPIC event rather than using Wait_on_Event. A call to Wait_on_Event causes the entire application to wait, thus blocking the progress of all the threads. By using the thread-oriented Event Management Services to handle the VMCPIC event, the same conditions (except for request IDs and user events signalled by Signal_User_Event) can be handled, while requiring at most one thread to wait.

Writing Multitasking Programs

A high-performance CPI Communications program may be structured to use the CMS multitasking facilities. Such a program would create multiple threads to handle operations asynchronously or to take advantage of the multiprocessor complex.

All CPI Communications calls, with the exception of the Wait_on_Event (XCWOE) call are thread-synchronous as opposed to virtual machine synchronous. This is to say, when a thread calls a CPI Communications routine that waits for data or enters some other type of wait, only that thread waits. No other threads are affected and another thread can be dispatched in place of the waiting thread.

When writing a multitasking CPI Communications program, keep in mind the following rules and multitasking concepts:

- All conversations are owned by the session and are accessible to all threads in the application. This means that one thread can allocate a conversation and others can send and receive data on it.
- While a CPI Communications call issued by a thread on a conversation is in progress, any calls issued by other threads to that same conversation are rejected. Note that the uncoordinated sharing of a conversation among multiple threads could lead to conversation state checks, especially when threads are running in a virtual multiprocessor virtual machine. If a conversation must be shared, use the multitasking synchronization primitives, such as mutexes or semaphores, to coordinate access to the conversation.
- Operations performed on two different conversations can be issued in parallel without interference. The only case in which thread synchronization could be necessary is if they are protected conversations in the same logical unit of work.
- To maintain concurrency, communications events should be waited for or trapped with the thread-oriented Event Management Services, rather than using the Wait_on_Event (XCWOE) interface. Wait_on_Event causes the entire application to wait, whereas the event management interface only requires a single thread to wait.

See the *z/VM: CMS Application Multitasking* for more information on CMS multitasking.

Summary of Common Routines

The following routines are part of IBM's SAA CPI Communications. These routines have common names and use a common syntax across all IBM environments that implement SAA CPI Communications. If you write application programs that call only these common routines, those applications will be more portable to other SAA environments.

The routines are shown here in alphabetic order by coded routine name. You must use the coded routine names when calling CPI Communications routines.

The *Common Programming Interface Communications Reference (https://publibfp.dhe.ibm.com/epubs/pdf/c2643999.pdf)* describes the syntax and parameters for each routine.

Table 70. Summary of CPI Communications Routines

Coded Routine Name	Routine Pseudonym	Routine Description
CMACCP	Accept_Conversation	Accepts an incoming conversation and initializes default values for various conversation characteristics. One or more of the default values may be overridden by calling the appropriate SET routine.
CMALLC	Allocate	Starts a basic or mapped conversation between the source program and the target program. If the target program is not within the same TSAF or CS collection, this routine allocates a session between the source and target LUs.
CMCFM	Confirm	Sends a confirmation request to the partner transaction program and waits for a reply. This routine lets the partner applications synchronize their processing.
CMCFMD	Confirmed	Sends a confirmation reply to the target transaction program. This is a positive response to the partner's call to the Confirm routine.
CMDEAL	Deallocate	Ends the conversation.
CMECS	Extract_Conversation_State	Extracts the conversation state value for the conversation and returns the value to the calling program.
CMECT	Extract_Conversation_Type	Extracts the conversation type value for the conversation and returns the value to the calling program.
CMEMN	Extract_Mode_Name	Extracts the mode name value for the conversation and returns the value to the calling program.
CMEPLN	Extract_Partner_LU_Name	Extracts the partner LU name value for the conversation and returns the value to the calling program.
CMESL	Extract_Sync_Level	Extracts the synchronization level value for the conversation and returns the value to the calling program.
CMFLUS	Flush	Flushes the calling program LU's send buffer for the conversation. The LU sends any information it has buffered for the conversation to the target LU.

Table 70. Summary of CPI Communications Routines (continued)

Coded Routine Name	Routine Pseudonym	Routine Description
CMINIT	Initialize_Conversation	Initializes various conversation characteristics from side information and default values. Conversation characteristics may be overridden by calling the appropriate SET routine.
CMPTR	Prepare_To_Receive	Changes the conversation from Send to Receive state in preparation to receive data.
CMRCV	Receive	Receives data, status, or both sent by the partner program for a mapped or basic conversation.
CMRTS	Request_To_Send	Notifies the partner program that the calling program is requesting to enter Send state for the conversation. The conversation state is changed only when the local program receives a send status from the partner program.
CMSCT	Set_Conversation_Type	Specifies whether the conversation is mapped or basic.
CMSDT	Set_Deallocate_Type	Specifies what kind of deallocation should be done for the conversation.
CMSED	Set_Error_Direction	Specifies the direction of data flow for the conversation for which the program detected an error.
CMSEND	Send_Data	Sends one data record to the partner transaction program when called for a mapped conversation. In this case, the data record consists entirely of data. Or, when called for a basic conversation, sends data to the partner transaction program. In this case, the data must consist of one or more APPC logical records.
CMSERR	Send_Error	Informs the partner program that the calling program has detected an error.
CMSF	Set_Fill	Specifies whether a program's next Receive call will get data independent of its logical record format. This routine applies only to basic conversations.
CMSLD	Set_Log_Data	Specifies log data for the conversation. Programs can use log data to help diagnose errors. This routine applies only to basic conversations.
CMSMN	Set_Mode_Name	Specifies the mode name for the conversation. The mode name denotes network properties for the session that contains the conversation.
CMSPLN	Set_Partner_LU_Name	Specifies the LU name where the target program is located.
CMSPTR	Set_Prepare_To_Receive_Type	Specifies the kind of prepare to receive to be done for the conversation.

Table 70. Summary of CPI Communications Routines (continued)

Coded Routine Name	Routine Pseudonym	Routine Description
CMSRC	Set_Return_Control	Sets the return control value for the conversation. This value specifies when the source LU is to return control to the local program when the Allocate routine is called, depending on session availability.
CMSRT	Set_Receive_Type	Specifies the kind of receive to be done for the conversation.
CMSSL	Set_Sync_Level	Sets the synchronization level for the conversation. This specifies the level of confirmation: NONE, CONFIRM, or SYNC POINT.
CMSST	Set_Send_Type	Sets the send type value for the conversation. The send type specifies what, if any, additional information is to be sent to the target program in addition to the data supplied with the Send routine, and whether the data is to be sent immediately or to be buffered.
CMSTPN	Set_TP_Name	Sets the TPN value for the conversation. TPN specifies the name of the target application program.
CMTRTS	Test_Request_To_Send_Received	Tests the specified conversation to see whether a Request_To_Send notification has been received from the partner program.

Summary of z/VM Extension Routines

These routines are individually described in detail in an appendix in the *Common Programming Interface Communications Reference (https://publibfp.dhe.ibm.com/epubs/pdf/c2643999.pdf)*. They are shown here in alphabetic order by coded routine name.

Table 71. Summary of z/VM Extension Routines

Coded Routine Name	Routine Pseudonym	Routine Description
XCECL	Extract_Conversation_LUWID	Extracts the SNA LU 6.2 architected Logical Unit of Work ID for a protected conversation.
XCECSU	Extract_Conversation_Security_User_ID	Extracts the access security user ID associated with a conversation.
XCECWU	Extract_Conversation_Workunit_ID	Extracts the CMS work unit ID for a conversation.
XCELFQ	Extract_Local_Fully_Qualified_ LU_Name	Extracts the local fully-qualified LU name for a conversation.
XCERFQ	Extract_Remote_Fully_Qualified_ LU_Name	Extracts the remote fully-qualified LU name for a conversation.

Table 71. Summary of z/VM Extension Routines (continued)

Coded Routine Name	Routine Pseudonym	Routine Description
XCETPN	Extract_TP_Name	Extracts the resolved TP name for a conversation.
XCIDRM	Identify_Resource_Manager	Declares to CMS a name (resource ID) by which the resource manager application will be known. For a local resource manager, this routine makes the name known to the system; for a global resource manager, this routine also makes the name known to the TSAF and CS collection.
XCSCUI	Set_Client_Security_User_ID	Lets an intermediate server specify an alternate user ID (the user ID of a specific client application).
XCSCSP	Set_Conversation_Security_Password	Sets the access security password value for the conversation. The security type must be PGM. The target LU uses this value and the user ID to verify the identity of the security requester.
XCSCST	Set_Conversation_Security_Type	Sets the security type (NONE, SAME, or PGM) for the conversation. The security type determines what security information is sent to the target. This lets the target verify the identity of the requester.
XCSCSU	Set_Conversation_Security_User_ID	Sets the security user ID value for the conversation. The security type must be PGM. The target LU uses this value and the security password to verify the identity of the requester.
XCSUE	Signal_User_Event	Queues an event to be reported by a subsequent Wait_on_Event call in the same virtual machine.
XCTRRM	Terminate_Resource_Manager	Ends ownership of a resource by a resource manager program.
XCWOE	Wait_on_Event	Waits on communications from one or more partners, and other events. Events reported are user events, allocation requests, information input, notification that a resource has been revoked, console input, and asynchronous Shared File System (SFS) requests.

Scenario 1: Request for a Global Resource

In the following example scenario, a z/VM user program requests a connection to a global resource manager program that is located in the same TSAF or CS collection. First, log on the server virtual machine and invoke the global resource manager program, then the user program needs to start a conversation with the global resource manager. After establishing the conversation, the user program

sends the name of a file to the global resource manager program. The global resource manager program then sends the entire contents of the file back to the user program. The user program then receives the file data and ends the conversation. Both programs use CPI Communications routines to communicate. For details on the routines shown here, refer to the *Common Programming Interface Communications Reference (https://publibfp.dhe.ibm.com/epubs/pdf/c2643999.pdf)*.

(A scenario for a local resource would be almost identical with the following scenario. The only difference: in the call to the Identify_Resource_Manager (XCIDRM) routine in step 1, the resource manager type would be specified as local instead of global.)

```
User Program                            Global Resource Manager Program
   (in Requester Virtual Machine)          (in Server Virtual Machine)
-----------------------------------     -------------------------------

                                        1. XCIDRM
                                        2. XCWOE
 3. CMINIT
 4. CMALLC
                                        5. If event=allocation request
                                           then CMACCP
 6. CMSST
 7. CMSEND
                                        8. if CMACCP okay then CMRCV
                                        9. CMSEND
10. CMRCV
        .                                       .
        .                                       .
                    send/receive sequence
        .                                       .
        .                                       .
                                       11. CMRCV
12. CMDEAL
                                       13. XCTRRM
```

Figure 95. Global Resource Request Scenario

Virtual Machine Preparation

- Both the requester and server virtual machines require the proper IUCV authorization.
- A CMS communications directory entry is not necessary in the requester virtual machine. When issuing the Initialize_Conversation routine, we will specify a symbolic destination name equal to the name of the global resource identified in the server virtual machine.

Program Functions

1. Identify the global resource manager to the z/VM system and the TSAF collection using the Identify_Resource_Manager (XCIDRM) routine.
2. Call the Wait_on_Event (XCWOE) routine to wait for an allocation request from the user program.
3. Initialize a conversation to the global resource manager program by calling the Initialize_Conversation (CMINIT) routine. The symbolic destination name used is the same as the resource ID specified in the XCIDRM call by the resource manager program.
4. Allocate the conversation from the user program to the resource manager program by calling the Allocate (CMALLC) routine.
5. The CMALLC from the user program completes the XCWOE call issued in step 2 with an allocation request event. Now, accept the inbound allocation request using the Accept_Conversation (CMACCP) routine.
6. Using the Set_Send_Type (CMSST) routine, set the *send_type* characteristic to CM_SEND_AND_PREPARE_TO_RECEIVE so that after the data is sent, our conversation will be in **Receive** state and we can receive data.
7. Send the name of the file we want from the resource manager program using the Send_Data (CMSEND) routine.

8. If CMACCP completes without error, receive the name of the file from the user program using the Receive (CMRCV) routine. Status should also be received indicating that the conversation is now in Send state for the resource manager.
9. Start the Send portion of the Send/Receive sequence. The resource manager program will issue as many Send_Datas (CMSENDs) as necessary to send the requested file to the user program.
10. Start the Receive portion of the Send/Receive sequence. The user program will issue as many Receives (CMRCVs) as necessary to receive the file sent by the resource manager program.
11. After sending the whole file to the user program, the resource manager program issues Receive (CMRCV) to put the user program in **Send** state and to receive data from the user program.

    After receiving the file contents successfully from the resource manager program, a CMRCV call will complete with a status received value indicating that we are now in **Send** state and that there is no more data coming from the resource manager program.
12. The user program calls the Deallocate (CMDEAL) routine to deallocate the conversation normally.
13. The deallocation by the user program completes the resource manager program's outstanding CMRCV with a CM_DEALLOCATED_NORMAL return code. Once this deallocation return code is reported, the resource manager program does not take any action, the conversation is ended. The resource manager program now terminates management of this resource by issuing the Terminate_Resource_Manager (XCTRRM) routine. (Note that instead of terminating the resource, the resource manager program could loop back to step 2 and wait for more allocation requests. In order to handle additional conversations, the resource manager program must have issued XCIDRM specifying that it will service more than one conversation.)

Scenario 2: Request for a Private Resource

In the following example scenario, a user program requests a private resource from a private resource manager program. The flows for this scenario are very similar to those in Scenario 1. First, the user program starts a conversation with the private resource manager program. After the conversation is set up, the user program sends the name of a file to the private resource manager program. The private resource manager program then sends the entire contents of the file back to the user program. The user program, after getting the file, ends the conversation. The programs use CPI Communications routines to communicate. For details on the routines shown here, see the *Common Programming Interface Communications Reference (https://publibfp.dhe.ibm.com/epubs/pdf/c2643999.pdf)*.

Note that for this scenario, both sides of the conversation use only the SAA CPI Communications routines, while Scenario 1, describing a global resource, requires the use of z/VM-extension routines.

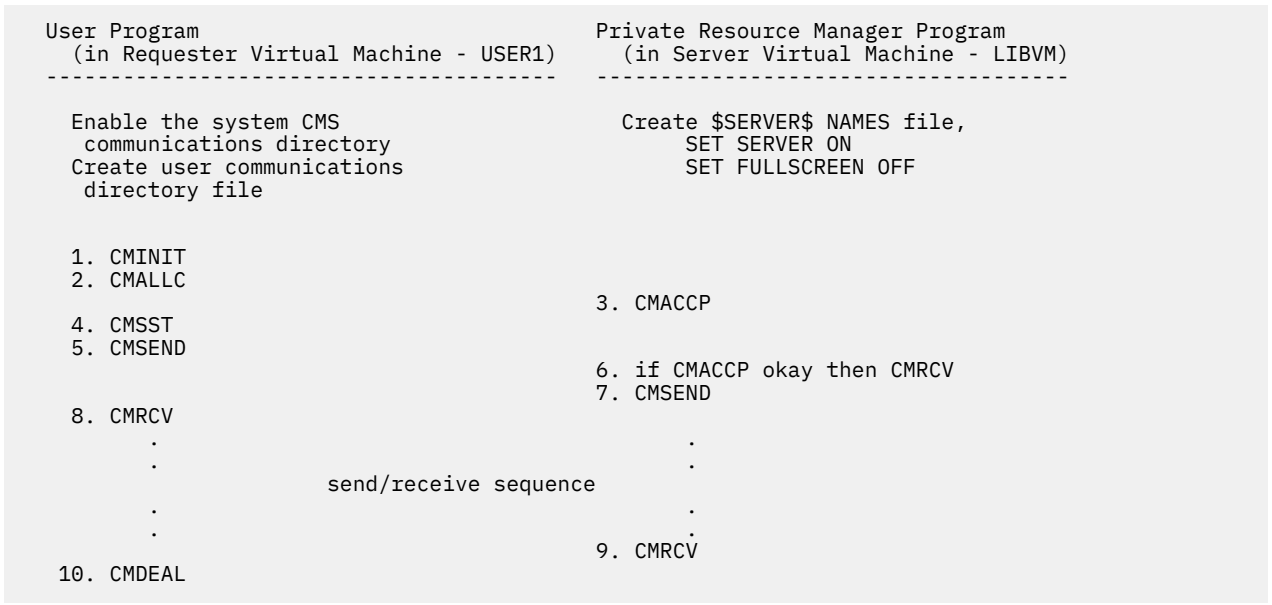


Figure 96. Private Resource Request Scenario

Virtual Machine Preparation

- For our scenario, the requester virtual machine has a user ID of USER1 and the private server virtual machine has a user ID of LIBVM,
- The requester virtual machine must be authorized to connect to LIBVM through the CP user directory.
- The CMS communications directory files need to be enabled in the requester virtual machine, USER1. Your system administrator usually would set up a system file called SCOMDIR NAMES, and that file would be in effect when you log on. You can also set up your own communications directory file and call it UCOMDIR NAMES. For this example, a communications directory file entry in UCOMDIR NAMES should be set up as follows:

```
:nick.BOOK      :tpn.LIBRARY
                :luname.*USERID LIBVM
                :security.SAME
```

  After setting up the UCOMDIR entry and issuing "SET COMDIR RELOAD", the user program can use BOOK (specified by the :nick. tag) as a symbolic destination name which will map to the TPN called LIBRARY (specified by the :tpn. tag) in the private server virtual machine that has user ID LIBVM (specified by the :luname. tag). Security is SAME (specified by the :security. tag), which means the access security user ID for the requester virtual machine is passed to the private server virtual machine.
- The private server virtual machine, LIBVM, must have a $SERVER$ NAMES file containing the names of private resources it is managing, and listing which virtual machines are authorized to access each resource.

  For purposes of this example, suppose the $SERVER$ NAMES file for this private server virtual machine just has a single entry as follows:

```
:nick.LIBRARY  :list.USER1
               :module.PRIVLIB
```

  Setting up the above entry in $SERVER$ NAMES will indicate that the access security user ID USER1 (specified by the :list. tag) is authorized to connect to the private resource LIBRARY (specified by the :nick. tag) and that the PRIVLIB exec (specified by the :module. tag) is to be invoked.

  In addition, SERVER must be set ON and FULLSCREEN must be set OFF or SUSPEND.

Program Functions

1. Initialize a conversation to the private resource manager program using the Initialize_Conversation (CMINIT) routine. The symbolic destination name specified on CMINIT should be BOOK which will map to the entry in the UCOMDIR NAMES file that we set up in the requester virtual machine.
2. Allocate the conversation to the private resource manager using the Allocate (CMALLC) routine.
3. The private resource request will go to the LIBVM virtual machine. CMS will look in its $SERVER$ NAMES to make sure that USER1 is authorized to connect to the private resource LIBRARY. Because USER1 is authorized, CMS then invokes the private resource manager program, PRIVLIB. The private resource manager program will then accept the conversation by using the Accept_Conversation (CMACCP) routine.

   Note: Because this is a private resource manager program and there is only one incoming conversation, there is no need to call the Identify_Resource_Manager (XCIDRM), Wait_on_Event (XCWOE) or Terminate_Resource_Manager (XCTRRM) routines. This allows the program to use only the SAA CPI Communications routines.
4. Call the Set_Send_Type (CMSST) routine to set *send_type* to CM_SEND_AND_PREPARE_TO_RECEIVE. The next time data is sent to the conversation partner, the conversation state will switch to Receive.
5. Send the name of the file we want from the resource manager program using the Send_Data (CMSEND) routine.

6. If the CMACCP completed okay, now receive the name of the file using the Receive (CMRCV) routine. Status should also be received indicating that the conversation is now in Send state for the resource manager.
7. Start the Send portion of the Send/Receive sequence. The resource manager program will issue as many Send_Datas (CMSENDs) as necessary to send the requested file to the user program.
8. Start the Receive portion of the Send/Receive sequence. The user program will issue as many Receives (CMRCVs) as necessary to receive the file sent by the resource manager program.
9. After sending the whole file to the user program, the resource manager program issues Receive (CMRCV) to put the user program in **Send** state and to receive data from the user program.
10. After receiving the file contents successfully from the resource manager program, a CMRCV call will complete with a status received value indicating that we are now in **Send** state and that there is no more data coming from the resource manager program. The user program calls the Deallocate (CMDEAL) routine to deallocate the conversation normally. This deallocate completes the resource manager program's outstanding CMRCV with a CM_DEALLOCATED_NORMAL return code, indicating that the conversation has been deallocated normally. Once this deallocate return code is reported, the resource manager program does not take any action, the conversation is ended.

Scenario 3: Synchronizing Multiple Updates

In the following example scenario, two applications work together to update three files. The applications use CPI Communications routines to communicate with each other and Resource Recovery routines to synchronize the file updates. The user program updates two SFS files and establishes a protected conversation with the target program. Then, the target program, which is invoked as a private resource, updates a third SFS file. After the target program updates the third SFS file, the target program asks if the updates should be committed or rolled back. If the target program commits the updates, then the user program asks if the updates should be committed or rolled back. If the user program commits the updates, then all updates are committed. If the target program or user program rolls back the updates after either prompt, all changes are rolled back.

Note: These example programs ask if the updates should be committed or rolled back. Most applications automatically proceed with whichever process is appropriate.

USERID1 is the user ID of the user virtual machine, which resides on SYSTEM1. USERID1 executes the user program, CRREXMP1 EXEC. CRREXMP1 EXEC updates the CHILDS LIST file in CRRDIR1 and the TOYSTORE ORDERS file in CRRDIR2.

USERID2 is the user ID of the target virtual machine, which may reside on either SYSTEM1 or SYSTEM2. USERID2 executes the target program, CRREXMP2 EXEC. CRREXMP2 EXEC updates the SANTAS SACK file in CRRDIR3.

Virtual Machine Preparation

The following information describes the user ID and virtual machine requirements needed to execute your programs. For more information on setting up a UCOMDIR NAMES file, $SERVER$ NAMES file, and an AVS virtual machine, see *z/VM: Connectivity*.

- USERID1 sets up a UCOMDIR NAMES file containing an entry for CRREXMP2. This UCOMDIR NAMES entry specifies information to connect to USERID2 and to start CRREXMP2 as a private resource.
- USERID2 uses the $SERVER$ NAMES file to identify the users that are authorized to use the private resource, CRREXMP2.
- USERID1 and USERID2 must have IUCV authorization.
- A recovery server virtual machine must be available on the system. If USERID1 and USERID2 are on different systems, a recovery server must be available on both systems.
- An SFS server virtual machine must be available to create SFS directories. If USERID1 and USERID2 are on different systems, an SFS server may be required on both systems. USERID1 and USERID2 must be enrolled in an SFS file pool with 500 blocks.

- If USERID1 and USERID2 reside on different systems, each system must have an AVS virtual machine available with private gateways (GATE1 and GATE2) established to support protected conversations.
- The private server virtual machine, USERID2, must have the following set:
  - SET SERVER ON
  - SET FULLSCREEN OFF
  - SET AUTOREAD OFF

Virtual Machines in the Same System

Assume that USERID1 and USERID2 are in the same system.

USERID1 sets up the following UCOMDIR NAMES entry for CRREXMP2:

```
:nick.CRREXMP2 :tpn.CRREXMP2
               :luname.*USERID USERID2
               :security.NONE
```

On the target virtual machine, the $SERVER$ NAMES entry for CRREXMP2 contains the following information:

```
:nick.CRREXMP2 :list.*
```

Virtual Machines on Different Systems

Now, assume that USERID1 and USERID2 are in separate systems in an SNA network. An AVS virtual machine must be available on SYSTEM1 and SYSTEM2.

The AVS virtual machine for SYSTEM1 defines a dedicated private gateway, called GATE1, that supports a protected conversation. The AVS virtual machine for SYSTEM2 defines a dedicated private gateway, called GATE2, that supports a protected conversation.

USERID1 sets up the following UCOMDIR NAMES entry for CRREXMP2:

```
:nick.CRREXMP2 :tpn.CRREXMP2
               :modename.FILESERV
               :luname.GATE1 GATE2
               :security.NONE
```

On the target virtual machine, the $SERVER$ NAMES entry for CRREXMP2 contains the following information:

```
:nick.CRREXMP2 :list.*
```

Overview for Synchronizing Multiple Updates

The following is a high-level overview of the REXX applications that update the files.

```
 CRREXMP1:                                          CRREXMP2:
   (in User Virtual Machine)                          (in Server Virtual Machine)
-----------------------------------                -------------------------------

Enable the user CMS communications                 Create $SERVER$ NAMES file,
  directory (UCOMDIR NAMES)                        SET SERVER ON,
                                                   SET FULLSCREEN OFF,
                                                   SET AUTOREAD OFF

 1. DMSSSPTO
 2. DMSSETAG

 3. DMSOPEN  (File 1)
 4. DMSWRITE (File 1)
 5. DMSCLOSE (File 1)

 6. DMSOPEN  (File 2)
 7. DMSWRITE (File 2)
 8. DMSCLOSE (File 2)

 9. CMINIT
10. CMSSL
11. CMALLC
                                                   12. CMACCP
13. CMCFM
                                                   14. CMRCV
                                                   15. CMCFMD
16. CMSEND
17. CMPTR
                                                   18. CMRCV
                                                   19. CMCFMD

                                                   20. DMSSSPTO
                                                   21. DMSSETAG

                                                   22. DMSOPEN  (File 3)
                                                   23. DMSWRITE (File 3)
                                                   24. DMSCLOSE (File 3)

< For backout processing >

                                                   25. SRRBACK
26. CMRCV
27. SRRBACK
                                                   28. CMSDT
                                                   29. CMDEAL
30. CMRCV

< For commit - commit processing >

                                                   31. CMDEAL
                                                   32. SRRCMIT
33  CMRCV
34. SRRCMIT

< For commit - backout processing >

                                                   35. CMDEAL
                                                   36. SRRCMIT
37. CMRCV
38. SRRBACK
                                                   39. CMSDT
                                                   40. CMDEAL
41. CMRCV
```

Figure 97. Synchronizing Multiple Updates Scenario

Program Functions

The following are step-by-step descriptions of the overview described previously. (1) is for CRREXMP1 and (2) is for CRREXMP2.

1. (1): DMSSSPTO to set synchronization point options
2. (1): DMSSETAG to set the transaction tag
3. (1): DMSOPEN the CHILDS LIST file in SFS directory .CRRDIR1

4. (1): DMSWRITE to CHILDS LIST
5. (1): DMSCLOSE with NOCOMMIT option to close CHILDS LIST
6. (1): DMSOPEN the TOYSTORE ORDERS file in SFS directory .CRRDIR2
7. (1): DMSWRITE to TOYSTORE ORDERS
8. (1): DMSCLOSE with NOCOMMIT option to close TOYSTORE ORDERS
9. (1): CMINIT (Initialize_Conversation) to set up the protected conversation
10. (1): CMSSL (Set_Sync_Level) to set the *sync_level* for the conversation to cm_sync_point
11. (1): CMALLC (Allocate) to allocate the protected conversation
12. (2): CMACCP (Accept_Conversation) to accept the protected conversation
13. (1): CMCFM (Confirm) to ensure partner has received allocation
14. (2): CMRCV (Receive) to get confirmation request
15. (2): CMCFMD (Confirmed) to respond to the confirmation request
16. (1): CMSEND (Send_Data) to send the data record to the partner
17. (1): CMPTR (Prepare_To_Receive) to enter Receive state
18. (2): CMRCV (Receive) to receive the data record and partner's request to enter Receive state
19. (2): CMCFMD (Confirmed) to confirm the partner's state request
20. (2): DMSSSPTO to set synchronization point options
21. (2): DMSSETAG to set the transaction tag
22. (2): DMSOPEN the SANTAS SACK file in SFS directory .CRRDIR3
23. (2): DMSWRITE to SANTAS SACK
24. (2): DMSCLOSE with NOCOMMIT option to close SANTAS SACK
25. (2): SRRBACK (Backout) to initiate backout processing
26. (1): CMRCV (Receive) to receive the backout indication
27. (1): SRRBACK (Backout) to perform backout processing
28. (2): CMSDT (Set_Deallocate_Type) to set the *deallocate_type* to CM_DEALLOCATE_ABEND
29. (2): CMDEAL (Deallocate) to deallocate the protected conversation
30. (1): CMRCV (Receive) to receive the deallocation notification
31. (2): CMDEAL (Deallocate) to deallocate the conversation after a successful syncpoint
32. (2): SRRCMIT (Commit) to initiate commit processing
33. (1): CMRCV (Receive) to receive the commit request indication
34. (1): SRRCMIT (Commit) to perform commit processing
35. (2): CMDEAL (Deallocate) to deallocate the conversation after a successful syncpoint
36. (2): SRRCMIT (Commit) to initiate commit processing
37. (1): CMRCV (Receive) to receive the commit request indication
38. (1): SRRBACK (Backout) to respond backout to commit request
39. (2): CMSDT (Set_Deallocate_Type) to set the *deallocate_type* to CM_DEALLOCATE_ABEND
40. (2): CMDEAL (Deallocate) to deallocate the protected conversation
41. (1): CMRCV (Receive) to receive the deallocation notification

Source Program for Synchronizing Multiple Updates

See “Example 3: Synchronizing Multiple Updates Using CRR and CPI Communications” on page 569 for the user and target REXX applications.

Scenario 4: Signaling a User Event

This example scenario demonstrates how the CPI Communications VM extension routines Signal_User_Event (XCSUE) and Wait_on_Event (XCWOE) can be used by an application to find out that a specific time interval has elapsed.

In addition to the communications partner programs written in REXX, an assembler program is needed to set the timer and to get control when the timer interrupt occurs. Here is a summary of what each program does:

SUESAMP1 EXEC
This is a REXX exec that allocates a conversation to another REXX exec, SUESAMP2 EXEC, with the intent of receiving data for 10 seconds. After allocating the conversation, SUESAMP1 performs a NUCXLOAD of the module created from the SUESAMP3 assembler program and then calls SUESAMP3. Next, SUESAMP1 calls Wait_on_Event in a loop to receive the data sent by SUESAMP2 and to determine when to stop receiving data and deallocate the conversation.

SUESAMP2 EXEC
This exec accepts the conversation from SUESAMP1 EXEC and sends data until that conversation is deallocated.

SUESAMP3 ASSEMBLE
This program is assembled into a relocatable module. It sets an interval timer using the OS/MVS simulated macro STIMER. When the interval elapses, STIMER drives the external interrupt handler in the SUESAMP3 module. When the interrupt handler gets control, it calls Signal_User_Event to post an event that can be reported by the Wait_on_Event called by SUESAMP1.

The comment sections of the program listings more fully describe what each program does. For the purposes of this example, USERID1 is the user ID of the virtual machine that will execute SUESAMP1 EXEC and the SUESAMP3 module. USERID2 is the user ID of the virtual machine that will execute SUESAMP2 EXEC.

Virtual Machine Preparation

This section describes the steps required to execute the programs used in this example scenario. For more information on setting up UCOMDIR NAMES and $SERVER$ NAMES files, see *z/VM: Connectivity*.

- USERID1 requires a UCOMDIR NAMES file with an entry for SUESAMP2, like this:

```
:nick.SUESAMP2  :tpn.SUESAMP2
                :luname.*USERID USERID2
```

  If USERID1 and USERID2 are not in the same TSAF collection, a `:modename.` tag is also required and the target gateway(s) must be listed on the `:luname.` tag.

  If a UCOMDIR NAMES file already existed, the `SET COMDIR RELOAD` command must be issued after the entry has been added. If a UCOMDIR NAMES file is created, the `SET COMDIR FILE USER UCOMDIR NAMES` command must be issued.

- USERID2 requires a $SERVER$ NAMES file that identifies the users that are authorized to use the private resource, SUESAMP2, like this:

```
:nick.SUESAMP2  :list.USERID1
```

  or like this, if all users are to be allowed access:

```
:nick.SUESAMP2  :list.*
```

- USERID2 should have an IUCV ALLOW statement in its CP directory unless connection access is restricted (see the *z/VM: Connectivity*). USERID2 must have the following commands in the PROFILE EXEC:
  - SET SERVER ON
  - SET FULLSCREEN OFF

  - SET AUTOREAD OFF
- After SUESAMP3 ASSEMBLE has been created, it must be compiled into a relocatable module on USERID1. Here is the sequence of commands:

```
GLOBAL TXTLIB CMSSAA VMLIB
GLOBAL MACLIB DMSGPI OSMACRO
ASSEMBLE SUESAMP3
LOAD SUESAMP3 (RLDSAVE)
GENMOD SUESAMP3 (SYSTEM)
```

After the execs have been created and all of the preceding steps have been completed, you can run the sample programs by entering:

```
suesamp1
```

from the command line of the USERID1 virtual machine.

The three programs and the output from executing them are listed on the following pages.

SUESAMP1 EXEC Listing

```
/*--------------------------------------------------------------------*/
/*  SUESAMP1 EXEC                                                     */
/*                                                                    */
/*  This EXEC is the source program for the demonstration of the      */
/*  CPI Communications VM extension routine Signal_User_Event         */
/*  (XCSUE).  It NUCXLOADs and calls SUESAMP3 MODULE (sample          */
/*  ASSEMBLE provided for this) to set a timer and handle the         */
/*  subsequent interrupt.  It establishes a CPI Communications        */
/*  conversation, and SUESAMP2 EXEC handles the partner side of       */
/*  the conversation.  A CMS communications directory entry for       */
/*  nickname SUESAMP2 is required by SUESAMP1.                        */
/*                                                                    */
/*  Because this program is for demonstration purposes, only          */
/*  minimal error checking is included.                               */
/*                                                                    */
/*  Main logic of SUESAMP1:                                           */
/*                                                                    */
/*     Initialize program constants                                   */
/*     Start a conversation with the partner                          */
/*     Execute nucleus extension program to set a timer for           */
/*        10 seconds.  Its interrupt handler, driven when the         */
/*        timer "pops", will issue Signal_User_Event (XCSUE).         */
/*     Do forever                                                     */
/*        Wait (XCWOE) for XCSUE user_event or data from partner      */
/*        If data is available, receive it                            */
/*           Else if a user_event occurred then leave                 */
/*     End                                                            */
/*     Show the results of the XCWOE (user_data from XCSUE)           */
/*     End the conversation with the partner                          */
/*--------------------------------------------------------------------*/

arg args

call Initialize
call Main_Prog

Get_Out:
   say
   Exit
/*--------------------------------------------------------------------*/
/*  INITIALIZE:  Set up program variables and constants               */
/*--------------------------------------------------------------------*/
Initialize:

   address command 'ESTATE CMREXX COPY *'
   if (rc ¬= 0) then call error 'CMREXX COPY file not found.'
      else do
         'execio * diskr CMREXX COPY * (finis stem PSEUDONYM.'
         do i = 1 to pseudonym.0
            interpret pseudonym.i
         end
      end
   say
```

```
   say '*'copies('-',  78)'*'
   msg = 'SUESAMP1:   CPI Communications XCSUE Sample Source Program'
   say '          ' msg
   say '*'copies('-',  78)'*'
   say

   return

/*--------------------------------------------------------------------*/
/*  MAIN_PROG:  Call Get_Conversation to start the conversation     */
/*              Call Set_Timer to start the timer program           */
/*              Do Forever                                          */
/*                 Call Wait_on_Event (XCWOE)                       */
/*                 If more data, call Receive_And_Confirmed         */
/*                    Else leave                                    */
/*              End                                                 */
/*              Show XCSUE user_data reported on XCWOE              */
/*              Call End_Conversation to end the conversation       */
/*--------------------------------------------------------------------*/
Main_Prog:

   call Get_Conversation
   call Set_Timer

   say 'XCWOE loop is beginning'
   loop_count = 0
   do forever
      x = 'XCWOE   -- Wait_on_Event'
      address cpicomm 'XCWOE res_id conv event_type'                ,
                      'event_info_len event_buf cm_rc'
      if (cm_rc ¬= cm_ok) then
         call error x, 'cm_rc', cm_return_code.cm_rc,               ,
                     cm_return_code.cm_ok
      if (event_type = xc_information_input) then call Receive_And_Confirmed
         else if (event_type = xc_user_event) then leave
            else call error x, 'event_type', xc_event_type.event_type, ,
                               xc_event_type.xc_user_event
   end

   say 'XCWOE loop is complete, performed' loop_count 'times'
   say
   say 'User_Event received by XCWOE:'
   say '   Event_ID parameter  =' res_id
   say '   User_Data parameter =' left(event_buf, event_info_len)
   say

   call End_Conversation

   return
/*--------------------------------------------------------------------*/
/*  SET_TIMER:  Nucxload SUESAMP3, which calls STIMER macro to set  */
/*              a timer for 10 seconds.  Its interrupt handler gets */
/*              control when the timer "pops" and issues XCSUE      */
/*--------------------------------------------------------------------*/
Set_Timer:

   'nucxload suesamp3 (system'
   'suesamp3'

   say 'SUESAMP3 called, timer set for 10 seconds'

   return

/*--------------------------------------------------------------------*/
/*  GET_CONVERSATION:  Issue CMINIT, CMSSL, CMALLC, and CMPTR to    */
/*                     start a CPI Communications conversation.     */
/*                     The CMINIT sets up conversation              */
/*                     characteristics and gives us the             */
/*                     conversation_id, the CMSSL sets the          */
/*                     sync_level of this conversation to           */
/*                     cm_confirm, and the CMALLC requests a        */
/*                     session for the conversation.  The CMPTR     */
/*                     will switch the conversation from send to    */
/*                     receive state, and also flow a confirmation  */
/*                     request to the partner since the sync_level  */
/*                     of the conversation is cm_confirm.           */
/*--------------------------------------------------------------------*/
Get_Conversation:

   x = 'CMINIT   -- Initialize_Conversation'
   sym_dest_name = 'SUESAMP2'
   address cpicomm 'CMINIT conv_id sym_dest_name cm_rc'
```

```
   if (cm_rc ¬= cm_ok) then                                    ,
      call error x, 'cm_rc', cm_return_code.cm_rc, cm_return_code.cm_ok

   x = 'CMSSL    -- Set_Sync_Level'
   sync_level = cm_confirm
   address cpicomm 'CMSSL conv_id sync_level cm_rc'
   if (cm_rc ¬= cm_ok) then                                    ,
      call error x, 'cm_rc', cm_return_code.cm_rc, cm_return_code.cm_ok

   x = 'CMALLC   -- Allocate'
   address cpicomm 'CMALLC conv_id cm_rc'
   if (cm_rc ¬= cm_ok) then                                    ,
      call error x, 'cm_rc', cm_return_code.cm_rc, cm_return_code.cm_ok
   x = 'CMPTR    -- Prepare_To_Receive'
   address cpicomm 'CMPTR  conv_id cm_rc'
   if (cm_rc ¬= cm_ok) then                                    ,
      call error x, 'cm_rc', cm_return_code.cm_rc, cm_return_code.cm_ok

   m = 'CMINIT, CMSSL, CMALLC, and CMPTR complete for conversation'
   say m conv_id

   return

/*--------------------------------------------------------------------*/
/*  RECEIVE_AND_CONFIRMED:  Issue CMRCV.  If confirm status is      */
/*                          received, then we have received a       */
/*                          complete data record so we'll respond   */
/*                          CMCFMD and increment the loop counter.  */
/*--------------------------------------------------------------------*/
Receive_And_Confirmed:

   reql = event_info_len

   x = 'CMRCV    -- Receive'
   address cpicomm 'CMRCV conv_id buf reql datr recl stat rts cm_rc'
   if (cm_rc ¬= cm_ok) then                                    ,
      call error x, 'cm_rc', cm_return_code.cm_rc, cm_return_code.cm_ok
   if (stat = cm_confirm_received) then do
      x = 'CMCFMD   -- Confirmed'
      address cpicomm 'CMCFMD conv_id cm_rc'
      if (cm_rc ¬= cm_ok) then                                 ,
         call error x, 'cm_rc', cm_return_code.cm_rc, cm_return_code.cm_ok
      loop_count = loop_count + 1
   end

   return

/*--------------------------------------------------------------------*/
/*  END_CONVERSATION:  Issue CMSERR to reject any data that might   */
/*                     have been sent but not received yet.  That   */
/*                     will switch the conversation to send state.  */
/*                     Issue CMSDT to set the deallocate_type to    */
/*                     cm_deallocate_flush, then CMDEAL to          */
/*                     deallocate the conversation.                 */
/*--------------------------------------------------------------------*/
End_Conversation:

   x = 'CMSERR   -- Send_Error'
   address cpicomm 'CMSERR conv_id rts_rec cm_rc'
   if (cm_rc ¬= cm_ok) then                                    ,
      call error x, 'cm_rc', cm_return_code.cm_rc, cm_return_code.cm_ok
   x = 'CMSDT    -- Set_Deallocate_Type'
   address cpicomm 'CMSDT conv_id cm_deallocate_flush cm_rc'
   if (cm_rc ¬= cm_ok) then                                    ,
      call error x, 'cm_rc', cm_return_code.cm_rc, cm_return_code.cm_ok

   x = 'CMDEAL   -- Deallocate'
   address cpicomm 'CMDEAL conv_id cm_rc'
   if (cm_rc ¬= cm_ok) then                                    ,
      call error x, 'cm_rc', cm_return_code.cm_rc, cm_return_code.cm_ok

   say 'CMSERR, CMSDT and CMDEAL complete for conversation' conv_id

   return
 *--------------------------------------------------------------------*/
/*  ERROR:  Display input error message and exit SUESAMP1.          */
/*--------------------------------------------------------------------*/
Error:

   parse arg message, parm_name, act_val, exp_val
   parm_msg = ' (Unexpected' parm_name 'value)'
```

```
   x = 'Error:'  message
   if (parm_name ¬= '') then x = x parm_msg

   say
   say x

   if (parm_name ¬= '') then do
      say
      say '   Expected value:' exp_val
      say '   Received value:' act_val
   end

   signal Get_Out

   return
```

SUESAMP2 EXEC Listing

```
/*--------------------------------------------------------------------*/
/*  SUESAMP2 EXEC                                                     */
/*                                                                    */
/*  This EXEC is the target program for the demonstration of the      */
/*  CPI Communications VM extension routine Signal_User_Event         */
/*  (XCSUE).  SUESAMP1 EXEC allocates a CPI Communications            */
/*  conversation, and this EXEC handles the partner side of that      */
/*  conversation.  An entry in the private resource authorization     */
/*  file ($SERVER$ NAMES) is required for SUESAMP2.                   */
/*                                                                    */
/*  Because this program is for demonstration purposes, only          */
/*  minimal error checking is included.                               */
/*                                                                    */
/*  Main logic of SUESAMP2:                                           */
/*                                                                    */
/*     Initialize program constants                                   */
/*     Start conversation initiated by the partner                    */
/*     Do forever                                                     */
/*        Send data (CMSEND) to the partner                           */
/*        If Send_Error indication received, then leave               */
/*     End                                                            */
/*     Receive conversation termination indication from partner       */
/*--------------------------------------------------------------------*/

arg args

call Initialize
call Main_Prog

Get_Out:
   say
   Exit

/*--------------------------------------------------------------------*/
/*  INITIALIZE:  Set up program variables and constants               */
/*--------------------------------------------------------------------*/
Initialize:

   address command 'ESTATE CMREXX COPY *'
   if (rc ¬= 0) then call error 'CMREXX COPY file not found.'
      else do
         'execio * diskr CMREXX COPY * (finis stem PSEUDONYM.'
         do i = 1 to pseudonym.0
            interpret pseudonym.i
         end
      end

   send_rec = 'This is a record of data for the CMSEND call.'
   send_rec_len = length(send_rec)
   say
   say '*'copies('-', 78)'*'
   msg = 'SUESAMP2:   CPI Communications XCSUE Sample Target Program'
   say '          ' msg
   say '*'copies('-', 78)'*'
   say

   return

/*--------------------------------------------------------------------*/
/*  MAIN_PROG:  Call Get_Conversation to start the conversation       */
/*              Do Forever                                            */
```

```
/*                 Call Send_And_Confirm to send data              */
/*                 If send_error was received then leave           */
/*              End                                                */
/*              Call End_Conversation to receive deallocation      */
/*-----------------------------------------------------------------*/
Main_Prog:

   call Get_Conversation     /* Do CMACCP, CMRCV, CMCFMD           */
   say 'CMSEND loop is beginning'
   loop_count = 0

   got_send_err = 0
   do forever
      call Send_and_Confirm
      if (got_send_err = 1) then leave
         else loop_count = loop_count + 1
   end

   say 'CMSEND loop is complete, performed' loop_count 'times'

   call End_Conversation

   return

/*-----------------------------------------------------------------*/
/*  GET_CONVERSATION:  Issue CMACCP, CMRCV, and CMCFMD to start a  */
/*                     CPI Communications conversation.  The CMACCP */
/*                     is in response to the source's CMALLC.  The  */
/*                     CMRCV will report the confirmation request   */
/*                     generated by the source's CMPTR (since the   */
/*                     sync_level of the conversation was set to    */
/*                     cm_confirm).  The CMCFMD will respond        */
/*                     positively to the confirmation request.  The */
/*                     CMSST specifies that a request for           */
/*                     confirmation will be sent with subsequent    */
/*                     CMSEND calls.                                */
/*-----------------------------------------------------------------*/
Get_Conversation:

   x = 'CMACCP   -- Accept_Conversation'
   address cpicomm 'CMACCP conv_id cm_rc'
   if (cm_rc ¬= cm_ok) then                                ,
      call error x, 'cm_rc', cm_return_code.cm_rc, cm_return_code.cm_ok
   x = 'CMRCV 1  -- Receive'
   reql = 2
   address cpicomm 'CMRCV conv_id buf reql datr recl stat rts cm_rc'
   if (cm_rc ¬= cm_ok) then                               ,
   call error x, 'cm_rc', cm_return_code.cm_rc, cm_return_code.cm_ok
   if (stat  ¬= cm_confirm_send_received) then            ,
      call error x, 'stat_rec', cm_status_received.stat,          ,
         cm_confirm_received

   x = 'CMCFMD   -- Confirmed'
   address cpicomm 'CMCFMD conv_id cm_rc'
   if (cm_rc ¬= cm_ok) then
   call error x, 'cm_rc', cm_return_code.cm_rc, cm_return_code.cm_ok

   x = 'CMSST    -- Set_Send_Type'
   address cpicomm 'CMSST conv_id cm_send_and_confirm cm_rc'
   if (cm_rc ¬= cm_ok) then
   call error x, 'cm_rc', cm_return_code.cm_rc, cm_return_code.cm_ok

   m = 'CMACCP, CMRCV, CMCFMD, and CMSST complete for conversation'
   say m conv_id

   return

/*-----------------------------------------------------------------*/
/*  SEND_AND_CONFIRM:  Issue CMSEND to send data to partner.       */
/*                     Because the send_type was previously set to  */
/*                     cm_send_and_confirm, a confirmation request  */
/*                     will be sent along with the data.  If a      */
/*                     send_error is received, a flag is set.       */
/*-----------------------------------------------------------------*/
Send_And_Confirm:

   x = 'CMSEND   -- Send_Data'
   address cpicomm 'CMSEND conv_id send_rec send_rec_len rts_rec cm_rc'

   if (cm_rc = cm_program_error_purging) then got_send_err = 1
      else if (cm_rc ¬= cm_ok) then                        ,
         call error x,'cm_rc', cm_return_code.cm_rc, cm_return_code.cm_ok
```

```
   return
/*-------------------------------------------------------------------*/
/*  END_CONVERSATION:  Issue CMRCV to receive the deallocation      */
/*                     notification to terminate the CPI            */
/*                     Communications conversation.                 */
/*-------------------------------------------------------------------*/

End_Conversation:

   x = 'CMRCV 2  -- Receive'
   reql = 2
   address cpicomm 'CMRCV conv_id buf reql datr recl stat rts cm_rc'
   if (cm_rc ¬= cm_deallocated_normal) then                  ,
      call error x, 'cm_rc', cm_return_code.cm_rc,                ,
                  cm_return_code.cm_deallocated_normal

   m = 'Normal deallocation received for conversation'
   say m conv_id

   return

/*-------------------------------------------------------------------*/
/*  ERROR:  Handle unexpected results                                */
/*-------------------------------------------------------------------*/
Error:

   parse arg message, parm_name, act_val, exp_val
   parm_msg = ' (Unexpected' parm_name 'value)'

   x = 'Error:'  message
   if (parm_name ¬= '') then x = x parm_msg

   say; say x

   if (parm_name ¬= '') then do
      say
      say '   Expected value:' exp_val
      say '   Received value:' act_val
   end

   signal Get_Out

   return
```

SUESAMP3 ASSEMBLE Listing

```
*-------------------------------------------------------------------*
*  SUESAMP3 ASSEMBLE                                                *
*                                                                   *
*  This program is the timer portion for the demonstration of the   *
*  CPI Communications VM extension routine Signal_User_Event        *
*  (XCSUE).  To create a module from this program, assemble it,     *
*  LOAD it with the RLDSAVE option, and GENMOD it with the SYSTEM   *
*  option.  Note that you will need to issue GLOBAL TXTLIB CMSSAA   *
*  prior to the LOAD.                                               *
*                                                                   *
*  Because this program is just for demonstration purposes, only    *
*  minimal error checking is included.                              *
*                                                                   *
*  Main Logic of SUESAMP3:                                          *
*                                                                   *
*     Issue call to STIMER macro to set a timer to "pop" in 10      *
*       seconds.                                                    *
*     When the timer "pops", the interrupt handler specified on the *
*       STIMER call is driven.  Signal_User_Event (XCSUE) is called *
*       in the interrupt handler so that we can post a user_event   *
*       to be reported on a subsequent Wait_On_Event (XCWOE) call   *
*       by our calling program (SUESAMP1).                          *
*-------------------------------------------------------------------*
*
SUESAMP3 CSECT                   Identify this program
         STM   R14,R12,12(R13)   Save system's registers
         LR    R12,R15           Set up addressability
         USING SUESAMP3,R12      Establish base register 12
         ST    R13,SAVEMAIN+4    Save pointer to system's save area
         LA    R13,SAVEMAIN      R13 points to our save area
*
*-------------------------------------------------------------------*
```

```
*  Issue STIMER macro to pop interrupt handler in 10 seconds.       *
*  Exit TIMESUP will get control when that time has expired.        *
*-------------------------------------------------------------------*
*
SETTIMER EQU   *
         STIMER REAL,TIMESUP,DINTVL=SECOND10
*
*-------------------------------------------------------------------*
*  Return control to CMS.                                           *
*-------------------------------------------------------------------*
*
EXIT     EQU   *
         L     R13,SAVEMAIN+4     Restore ptr. to system's save area
         L     R14,12(R13)        Restore the system registers
         LM    R0,R12,20(R13)     Get registers
         BR    R14                Return control to the system
**-----------------------------------------------------------------*
*  The following routine is the external interrupt handler for      *
*  STIMER interrupts.  Save the registers, issue XCSUE to signal    *
*  that the timer exit was issued, restore registers and return.    *
*-------------------------------------------------------------------*
*
TIMESUP  EQU   *
         STM   R14,R12,12(R13)    Save system's registers
         LR    R12,R15            Set up addressability
         USING TIMESUP,R12        Establish base register 12
*
         LA    R0,SUELNGTH        Obtain storage needed for interrupt
         CMSSTOR OBTAIN,BYTES=(R0)     handler
         LR    R5,R1              R1 has address of storage obtained
         USING SUE,R5             Map our storage to SUE DSECT
*
         ST    R13,SUESAVE+4      Save pointer to system's save area
         LA    R13,SUESAVE        R13 points to our save area
*
*                                 Call XCSUE to post the user event
         CALL  XCSUE,(EVENTID,USERDATA,USERDLEN,SUERC),VL
*
         L     R13,SUESAVE+4      Restore R13 pointer
*
         LA    R0,SUELNGTH        Release storage we obtained
         CMSSTOR RELEASE,BYTES=(R0),ADDR=(R5)
         DROP R5
*
         LM    R14,R12,12(R13)    Restore the system's registers
         BR    R14                Return to CMS
*
*-------------------------------------------------------------------*
*   Program storage areas and constants.                            *
*-------------------------------------------------------------------*
*
SAVEMAIN DS    18F                Save area for the user program
         DS    0D                 Need doubleword boundary
*                  HHMMSSTH       STIMER DINTVL format (10 seconds)
SECOND10 DC    CL8'00001000'
EVENTID  DC    CL8'SUESAMP3'      Event_ID for XCSUE
*                                 User_Data for XCSUE
USERDATA DC    CL31'Timer has expired for SUESAMP3!'
         DS    0F                 Fullword alignment
USERDLEN DC    A(L'USERDATA)      User_Data length for XCSUE
SUERC    DC    F'0'               Return_Code for XCSUE
**-----------------------------------------------------------------*
*   DSECTS                                                          *
*-------------------------------------------------------------------*
*
SUE      DSECT
SUESAVE  DS    18F                Save area for the user program
SUELNGTH EQU   *-SUE              Length of DSECT
*
         REGEQU                   Include the register equates
         END   SUESAMP3
```

Execution Results

The results of executing SUESAMP1 EXEC are shown in the following figures. The number of times the loop is executed may vary from execution to execution.

Allocating Program's Results

```
*-------------------------------------------------------------------------*
         SUESAMP1:   CPI Communications XCSUE Sample Source Program
*-------------------------------------------------------------------------*

CMINIT, CMSSL, CMALLC, and CMPTR complete for conversation 00000000
SUESAMP3 called, timer set for 10 seconds
XCWOE loop is beginning
XCWOE loop is complete, performed 2063 times

User_Event received by XCWOE:
  Event_ID parameter  = SUESAMP3
  User_Data parameter = Timer has expired for SUESAMP3!

CMSERR, CMSDT and CMDEAL complete for conversation 00000000

Ready;
```

Figure 98. Results on USERID1 Virtual Machine

Accepting Program's Results

```
*-------------------------------------------------------------------------*
         SUESAMP2:   CPI Communications XCSUE Sample Target Program
*-------------------------------------------------------------------------*

CMACCP, CMRCV, CMCFMD, and CMSST complete for conversation 00000000
CMSEND loop is beginning
CMSEND loop is complete, performed 2063 times
Normal deallocation received for conversation 00000000

Ready;
```

Figure 99. Results on USERID2 Virtual Machine

Scenario 5: Using the VMCPIC Event

- **Example 1 – Replacing XCWOE**

  A multitasking application should use Event Management Services instead of Wait_on_Event (XCWOE) to wait on communications events so that only one thread, rather than the entire application, is required to wait.

```
Using XCWOE                  Using VMCPIC
-----------                 ----------------
     .                       .
     .                       .
     .                       .
XCWOE(....)                 EventMonitorCreate(for VMCPIC
                            event with key *
                            or VMCONINPUT event)
                            returns a monitor token

                            EventWait(on monitor token
                            returned above)

                            EventRetrieve(on monitor token
                            returned above)
     .                       .
     .                       .
     .                       .
```

Figure 100. Replacing XCWOE

- **Example 2 – Allocation requests on any resource**

  The following application waits until any allocation request arrives for any resource or until a timeout timer has expired. It uses Timer Services for the timeout timer.

```
Explanation                          Application
-----------                          -----------
         .                               .
         .                               .
         .                               .
1) Identified as resource            XCIDRM(for resource SOMENAME)
   manager for resource
   SOMENAME
         .                               .
         .                               .
         .                               .
2) Start a timer for the timeout     TimerStartInt(...)
                                       returns timer token TOKN

3) Build event key X'00000001'*      eventkey =
   using pseudonym for event_type      XC_ALLOCATION_REQUEST'*'

4) Create a monitor for the          EventMonitorCreate(for
   events on which to wait              VMTIMER event with key TOKN*
                                        or VMCPIC event with key eventkey)
                                        returns a monitor token
         .                               .
         .                               .
         .                               .

5) Wait for either the timeout       EventWait(on above monitor token)
   timer to expire or an               application goes into a wait
   allocation request on any
   resource
           .                                .
           .                                .
           .                                .
6) One of the following signals      application wakes up and continues
   occurs                            processing
   - an allocation request for
     SOMENAME                              .
   - timeout timer expires or is           .
     stopped                               .

7) Get event data associated         EventRetrieve(on monitor token
   with signal                       returned above)
           .                             .
           .                             .
           .                             .
```

Figure 101. Allocation Requests on Any Resource

- **Example 3 – Resource revoked notification on any resource**

  The following application waits until any resource revoked notification comes in for any resource or until its timeout timer has expired. To do this, it needs to change the VMCPIC event key on the EventMonitorCreate in step 3 above. This will be done using the CPI Communications pseudonym for the *event_type*. (In step 6 the application wakes up due to a resource revoked notification rather than an allocation request). Our new step 3 would look like this:

```
Explanation                          Application
-----------                          -----------
         .                               .
         .                               .
         .                               .
3) Build event key X'00000003'*      eventkey =
   using pseudonym for event_type        XC_RESOURCE_REVOKED'*'
   Create a monitor for the          EventMonitorCreate(for
   events on which to wait              VMTIMER event with key TOKN*
                                        or VMCPIC event with key eventkey)
                                        returns a monitor token
         .                               .
         .                               .
         .                               .
```

Figure 102. Resource Revoked Notification on Any Resource

- **Example 4 – Information input on a conversation**

  The following application waits for information input on a particular conversation. In this example we will deal with one event.

```
Explanation                          Application
-----------                          -----------
         .                                .
         .                                .
         .                                .
1) Accept a conversation and get     CMACCP(..)
   the conversation_ID                 returns conversation_ID 00000001
         .                                .
         .                                .
         .                                .

2) Build event key of event type     eventkey =
   concatenated with the                 XC_INFORMATION_INPUT conversation_ID
   conversation_ID

3) Create a monitor for the          EventMonitorCreate(for
   event on which to wait               VMCPIC event with key
                                        eventkey)
                                        returns a monitor token

4) Wait for information input        EventWait(on above monitor token)
   on conversation 00000001            application goes into a wait
         .                                .
         .                                .
         .                                .
5) Information input occurs          application wakes up
   on conversation 00000001

6) Get the event data which          EventRetrieve(on above monitor
   will give you the                   token)
   event_info_length

7) Receive information from partner  CMRCV(...)
         .                                .
         .                                .
         .                                .
```

Figure 103. Information Input on a Conversation

- **Example 5 – Console input**

  The following application waits for console input only.

```
Explanation                          Application
-----------                          -----------
         .                                .
         .                                .
         .                                .

1) Create a monitor for the          EventMonitorCreate(for
   event on which to wait               VMCONINPUT event)
                                        returns a monitor token

2) Wait for console input            EventWait(on above monitor token)
                                       application goes into a wait
         .                                .
         .                                .
         .                                .
3) Console input occurs              application wakes up

4) Read the input from the           LINERD(...)
   console

5) Handle input as appropriate
         .                                .
         .                                .
         .                                .
```

Appendix A. Assembler Examples

These Assembler examples follow:

Example 1: Assembler Application Using the CSL Extract/Replace Routine

The following assembler program, called CSLASSEM ASSEMBLE, calls DMSERP, the CSL routine in VMLIB that accesses the extract/replace function. This particular call is extracting the access mode of the first read/only CMS disk.

Assembler Application Using Extract/Replace

```
* ================================================================ *
* This program issues a call to the EXTRACT/REPLACE CSL routine,   *
* DMSERP, to extract the accessmode of the first READ/ONLY CMS disk.  *
* ================================================================ *
*
CSLASSEM CSECT ,                       Program identifier
MAINENT  DS    0H
         USING *,R15
         B     PROLOG
         DC    AL1(16)
         DC    C'CSLASSEM  90.058'
         DROP  R15
PROLOG   STM   R14,R12,12(R13)          Standard linkage
         LR    R12,R15
PSTART   EQU   CSLASSEM
         USING PSTART,R12
         ST    R13,SAVE01+4
         LA    R14,SAVE01
         ST    R14,8(,R13)
         LR    R13,R14
*
*   CALL EXTRACT/REPLACE VIA CSL
*
* ================================================================ *
*
* An EXTRN statement is need to identify to the ASSEMBLER that DMSCSL
* is an EXTeRNal reference.
*
         EXTRN DMSCSL
*
* The parameters on the call to DMSCSL are described in the DATA
* section of this program.
*
* The call to EXTRACT/REPLACE is issued to determine what the
* access mode (INFONAME) of the first READ/ONLY disk (SARGNAM).
* BUFFER contains the access mode of the first READ/ONLY disk.
*
         CALL DMSCSL,(EXTREP,RTNCODE,DOEXTRAT,NUMARGS,                *
               INFONAME,BUFFER,DATATYP,BUFLEN,                        *
               FLAGS,SRCHTYP,TOKEN,                                   *
               SARGNAM,SARGVAL,SVALTYP,SVALLEN,SARGTYP),VL
*
*   Display results using the APPLMSG macro
*
* ================================================================ *
*
         APPLMSG APPLID=TCP,HEADER=NO,TEXT='RTNCODE= &&1',            *
               SUB=(DECA,(RTNCODE,4))
         APPLMSG APPLID=TCP,HEADER=NO,TEXT='BUFFER= &&1',             *
               SUB=(HEXA,(BUFFER,20))
         APPLMSG APPLID=TCP,HEADER=NO,TEXT='BUFFER= &&1',             *
               SUB=(CHARA,(BUFFER,8))
         APPLMSG APPLID=TCP,HEADER=NO,TEXT='DATATYP = &&1',           *
               SUB=(DECA,(DATATYP,4))
```

```
         APPLMSG APPLID=TCP,HEADER=NO,TEXT='BUFLEN = &&1',            *
               SUB=(DECA,(BUFLEN,4))
*RETURN CODE(RC);
         L     R15,RC
         L     R13,4(,R13)
         L     R14,12(,R13)
         LM    R00,R12,20(R13)
         BR    R14
*END SAMPERXP
* ================================================================== *
* ============           DATA Section                ============== *
* ================================================================== *
*
DATA     DS    0H
         DS    0F
SAVE01   DS    18F               Save area used for linkage
         DS    0F
WORDONE  DC    F'1'              Define a value of one
         DS    0D
EXTREP   DC    CL8'DMSERP  '             The CSL routine to be invoked
RTNCODE  DC    F'99'                     Setup bad return code
*                                        when function is complete it
*                                        should be zero (0).
DOEXTRAT DC    CL8'EXTRACT '             Identify function - EXTRACT
NUMARGS  DC    F'1'                      Number of arguments in call
*                                        There is only one in
*                                         this case
*
* Infoname identifies the information that is to be extracted.
INFONAME DC    CL20'ACCESS_MODE         '
*
*  Buffer contains the value of the extracted data.
*
BUFFER   DC    CL1'0'
*
* Data type identifies the type of data being extracted.
* The value of DATATYP changes after the call to DMSCSL.
*      32 -character string
*       4 -numeric
*       9 -indicator
*      13 -address
*
DATATYP  DC    F'0'
*
* Buflen contains the length of the buffer on input to the CSL call.
* On output the BUFLEN contains the actual length of the data being
* extracted.
BUFLEN   DC    F'4'
*
* FLAGS are used to control the search process
FLAGS    DC    CL8'00000000'
*
* SRCHTYP is a keyword  either AND or OR indicating the way multiple
* search arguments will be combined.
SRCHTYP  DC    CL4'OR  '
* Token is used for storing input and output information when
* multiple occurrences of the designated INFONAME exist.
*
TOKEN    DS    F
*
*  The next 5 parameters comprise the SEARCH ARGUMENT.
* SARGNAM contains an information name that is used to qualify the
*         search for INFONAME
* SARGVAL is the value which the SARGNAM value will be compared against
* SVALTYP contains the data type of the value contained in SARGVAL
* SVALLEN is the length, in bytes, of the value contained in SARGVAL.
* SARGTYP is the type of comparison to be preformed.  Valid values:
*             EQ -equal
*             GT -greater than
*             LT -less than
*             GE -greater or equal
*             LE -less or equal
*             NE -not equal
SARGNAM  DC    CL20'CMS_READ_ONLY_DISK  '
SARGVAL  DC    CL1'1'
SVALTYP  DC    F'9'
SVALLEN  DC    F'1'
SARGTYP  DC    CL2'EQ'
*
RC       DC    F'0'
R00      EQU   0
R01      EQU   1
```

```
R02      EQU   2
R03      EQU   3
R04      EQU   4
R05      EQU   5
R06      EQU   6
R07      EQU   7
R08      EQU   8
R09      EQU   9
R10      EQU   10
R11      EQU   11
R12      EQU   12
R13      EQU   13
R14      EQU   14
R15      EQU   15
         DS    0D
@ENDDATA EQU   *
@MODLEN  EQU   @ENDDATA-CSLASSEM
         END   CSLASSEM
```

After executing the above program, here is what would be displayed on your terminal (assuming that the B disk is the first minidisk accessed as read/write):

```
RTNCODE= 0
BUFFER= C2000000
BUFFER= B
DATATYP = 32
BUFLEN = 1
```

Remember when writing an assembler program with a call to a CSL routine:

- Use the following statement to identify to the assembler that DMSCSL is an external reference:

```
EXTRN DMSCSL
```

Example 2: Assembler Application Using CSL Routines to Open, Read, and Close Files

The following assembler program, called OPRDCL, opens a file, called TESTMC FILE, reads one record from the file, and closes the file. This application assumes that TESTMC FILE is in a directory accessed as B. OPRDCL ASSEMBLE contains the following.

```
OPRDCL  CSECT
         USING *,R15
         STM   R14,R12,12(R13)
         L     R12,=A(SAVE01)
         ST    R12,8(,R13)
         ST    R13,4(,R12)
         LR    R13,R12
         BALR  R12,0
         USING *,R12
         DROP  R15
         SPACE 3
         APPLMSG TEXT='DOING DMSOPEN'
         CALL  DMSCSL,(ROUTINE,RETURN,REASON,PARM1,PARM2,              -
               PARM3,                                                  -
               PARM4,                                                  -
               PARM5),VL
         BAL   8,DISPLAY
         APPLMSG TEXT='DOING DMSREAD'
         CALL  DMSCSL,(ROUTIN1,RETURN,REASON,PARM5,PARM7,PARM8,        -
               PARM9,                                                  -
               PARM10,                                                 -
               PARM11),VL
         BAL   8,DISPLAY
         SPACE 3
         APPLMSG TEXT='DOING DMSCLOSE'
         CALL  DMSCSL,(ROUTIN2,RETURN,REASON,PARM5,PARM12,             -
               PARM13),VL
         BAL   8,DISPLAY
         B     EXIT
         SPACE 3

DISPLAY  L     2,RETURN
         APPLMSG TEXT='RETURN IS &&1',SUB=(HEX,(2))
```

```
         L     R4,REASON
         APPLMSG TEXT='REASON CODE &&1',SUB=(HEX,(4))
         SPACE 3
         BR    R8                          LEAVE!
*********************************************************************
EXIT     L     R13,=A(SAVE01)
         L     R13,4(,R13)
         ST    R15,16(R13)
         LM    R14,R12,12(R13)
         BR    R14
         EJECT
SAVE01   DS    18F
ROUTINE  DC    C'DMSOPEN '
ROUTIN1  DC    C'DMSREAD '
ROUTIN2  DC    C'DMSCLOSE'
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'        '
PARM6     DC    C'        '
PARM7     DC    F'2'
PARM8     DC    F'200'
PARM9     DC    CL200' '
PARM10    DC    F'200'
PARM11    DC    F'0'
PARM12    DC    C'COMMIT'
PARM13    DC    A(L'PARM12)
         SPACE 4
R0       EQU   0
R1       EQU   1
R2       EQU   2
R3       EQU   3
R4       EQU   4
R5       EQU   5
R6       EQU   6
R7       EQU   7
R8       EQU   8
R9       EQU   9
R10      EQU   10
R11      EQU   11
R12      EQU   12
R13      EQU   13
R14      EQU   14
R15      EQU   15
         EJECT
         END
```

If the file was successfully opened, read, and closed, the following is displayed:

```
DOING DMSOPEN
RETURN IS 00000000.
REASON CODE 00000000.
DOING DMSREAD
RETURN IS 00000000.
REASON CODE 00000000.
DOING DMSCLOSE
RETURN IS 00000000.
REASON CODE 00000000.
```

Appendix B. C Example

The following C application, called CSLC C, calls DMSERP, the CSL routine in VMLIB that accesses the extract/replace function. This particular call is extracting the access mode of the first read-only CMS disk. CSLC C contains the following:

```
#include <stdio.h>

 /* This sample C application program calls Extract/Replace,   */
 /* via DMSCSL, to obtain the access mode of the first         */
 /* read/only CMS disk.                                        */

 /* DMSCSL   - External interface routine to extract/replace   */
 /* RTNCODE  - Return code from extract/replace                */
 /* BUFFER   - Value that was extracted                        */
 /* BUFF     - Value that was extracted                        */
 /* DATATYPE - Data type of data extracted                     */
 /* BUFLNGTH - Length of buffer/length of extracted data       */
 /* SRCHTYPE - Logical type of search                          */
 /* TOKEN    - Pointer of sorts                                */
 /* SARGNAM1 - Search argument name                            */
 /* SARGVAL1 - Search argument value                           */

     #pragma linkage(DMSCSL,OS)

     extern int       DMSCSL(const char *RTNNAME, int *RC, ...);

  main()
   {
            int       RTNCODE = 1;
            char      BUFFER[] = "XXXXX";
            int       DATATYPE = 0;
            int       BUFLNGTH = 5;
            char      SRCHTYPE[] = "OR  ";
            int       TOKEN;
            char      SARGNAM1[] = "CMS_READ_ONLY_DISK  ";
            char      SARGVAL1 = '1';
            int       i;     printf ("Before RESET call:\n");
     printf ("RTNCODE = %d\n",RTNCODE);
     printf ("BUFFER = ");
     for(i = 0; i < 5;i++)
        printf("%1x",BUFFER[i]);
     printf("\n");
     printf ("DATATYPE = %d\n",DATATYPE);
     printf ("BUFLNGTH = %d\n",BUFLNGTH);
     printf ("SRCHTYPE = \"%s\"\n",SRCHTYPE);
     printf ("TOKEN    = %d\n",TOKEN);
     printf ("SARGNAM1 = \"%s\"\n",SARGNAM1);
     printf ("SARGVAL1 = \"%c\"\n",SARGVAL1);

     DMSCSL("DMSERP  ", &RTNCODE, "RESET   ");

     printf("\n");

     printf ("After RESET call:\n");
     printf ("RTNCODE = %d\n",RTNCODE);
     DMSCSL("DMSERP  ", &RTNCODE, "EXTRACT ", 1,
             "ACCESS_MODE         ", &BUFFER, &DATATYPE, &BUFLNGTH,
             "00000000", &SRCHTYPE, &TOKEN,
             &SARGNAM1, &SARGVAL1, 9, 1, "EQ");


     printf("\n");

     printf ("After EXTRACT call:\n");
     printf ("RTNCODE = %d\n",RTNCODE);
     printf ("BUFFER = ");
     for(i = 0; i < 5;i++)
        printf("%1x",BUFFER[i]);
     printf("\n");
     printf ("DATATYPE = %d\n",DATATYPE);
     printf ("BUFLNGTH = %d\n",BUFLNGTH);
     printf ("SRCHTYPE = \"%s\"\n",SRCHTYPE);
     printf ("TOKEN    = %d\n",TOKEN);
     printf ("SARGNAM1 = \"%s\"\n",SARGNAM1);
```

```
      printf ("SARGVAL1 = \"%c\"\n",SARGVAL1);


      if(RTNCODE == 0)
        {
         printf("\nYour first accessed R/O mode is ");
         for(i = 0; i < BUFLNGTH;i++)
            printf("%1c",BUFFER[i]);
         printf("\n");
        {
```

Remember the following notes when coding a C program with a call to a CSL routine:

- To declare DMSCSL, use the "extern" statement as shown above.
- When passing an integer value as a parameter, you must preface the parameter name with an ampersand (&) as shown in the DMSCSL calls in the above program.
- To pass a parameter as a literal, you must surround it with double quotation marks rather than single quotation marks.
- CSL does not append a null on character strings it returns. The C program must take this into consideration.
- The #pragma statement for DMSCSL, as shown in the above program, is required in order to ensure that DMSCSL is called with the correct linkage.

Appendix C. COBOL Examples

These COBOL examples follow:

Example 1: Simple COBOL Application

The following is a sample COBOL application called WELCOME. This application prompts you to enter your first name and last name. Then it welcomes you to CMS. WELCOME COBOL contains the following code:

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. MYPROG.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       77  FNAME         PIC X(22) VALUE "ENTER YOUR FIRST NAME:".
       77  LNAME         PIC X(23) VALUE "AND NOW YOUR LAST NAME:".
       01  ANSWR.
           05    ANSLT   PIC X(16) VALUE "WELCOME TO CMS, ".
           05    AFRST   PIC X(8)  VALUE SPACES.
           05    FILLER  PIC X     VALUE SPACES.
           05    ALAST   PIC X(8)  VALUE SPACES.
       PROCEDURE DIVISION.
           DISPLAY FNAME UPON CONSOLE.
           ACCEPT  AFRST FROM CONSOLE.
           DISPLAY LNAME UPON CONSOLE.
           ACCEPT  ALAST FROM CONSOLE.
           DISPLAY ANSWR UPON CONSOLE.
           STOP RUN.
```

Figure 104. Simple COBOL Application

Example 2: Complete COBOL Application

The following COBOL application lets you add, change, delete, or display records of peoples' names, by serial number. The records must be added before they can be changed, deleted, or displayed.

This application package consists of two parts:

- The source COBOL program, called COBOL1 COBOL, prompts you for information. Then it adds, changes, deletes, or displays the records of the work file.
- The exec program, called DRIVE1 EXEC. DRIVE1 invokes COBOL1. It does the file management for COBOL1 using CMS commands. COBOL1 creates a temporary work file, so DRIVE1 checks if the file already exists. If the work file exists, DRIVE1 issues an error message and does not call COBOL1. If the work file does not exist, DRIVE1 issues the FILEDEF commands and calls COBOL1.

  Upon return, DRIVE1 tests the return code set by COBOL1. If the return code indicates an incomplete work file, DRIVE1 erases the work file. If the return code indicates a completed work file, DRIVE1 erases the old master file and renames the work file as the new master file. The master file is called NAMES DATA.

To execute this application, issue the following commands:

```
COBOL2 COBOL1
GLOBAL TXTLIB VSC2LTXT
LOAD COBOL1
GENMOD COBOL1
DRIVE1
```

DRIVE1 EXEC contains the following information:

```
&TRACE
STATE WORK DATA A
&IF &RETCODE GT 0 &GOTO -OK
&TYPE FILE 'WORK DATA A' EXISTS.  ERASE AND TRY AGAIN.
&EXIT
-OK
FILEDEF NAMES DISK NAMES DATA
FILEDEF WORK DISK WORK DATA
COBOL1
&IF &RETCODE NE 0 &GOTO -NG
STATE WORK DATA A
&IF &RETCODE GT 0 &GOTO -NF
ERASE NAMES DATA A
RENAME WORK DATA A NAMES DATA A
-NF
&EXIT
-NG
ERASE WORK DATA A
&EXIT
```

COBOL1 COBOL contains the following information:

Complete COBOL Application

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. COBOL1.
       ENVIRONMENT DIVISION.
       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
           SELECT INFILE ASSIGN TO DA-3390-S-NAMES
           ACCESS MODE IS SEQUENTIAL.
           SELECT OUTFILE ASSIGN TO DA-3390-S-WORK
           ACCESS MODE IS SEQUENTIAL.
       DATA DIVISION.
       FILE SECTION.
       FD  INFILE
           RECORDING MODE IS F
           LABEL RECORDS OMITTED
           DATA RECORD IS EMPRECIN.
       01  EMPRECIN.
           03   SERIALNIN   PIC X(6).
           03   FRSTNMIN    PIC X(16).
           03   LASTNMIN    PIC X(16).
       FD  OUTFILE
           RECORDING MODE IS F
           LABEL RECORDS OMITTED
           DATA RECORD IS EMPRECOUT.
       01  EMPRECOUT.
           03   SERIALNOUT  PIC X(6).
           03   FRSTNMOUT   PIC X(16).
           03   LASTNMOUT   PIC X(16).
       WORKING-STORAGE SECTION.
       01  FNAME     PIC X(16)  VALUE SPACES.
       01  LNAME     PIC X(16)  VALUE SPACES.
       01  INPLINE1.
           03   FNTYPE    PIC X        VALUE SPACES.
           03   FILLER    PIC X.
           03   EMPSER    PIC X(6)   VALUE SPACES.
       01  ERRMSG    PIC X(20)  VALUE "INCORRECT SERIAL NO.".
       01  GOODMSG   PIC X(20)  VALUE "OPERATION COMPLETED.".
       01  MENULINE1 PIC X(21)  VALUE "ENTER FUNCTION NUMBER".
       01  MENULINE2 PIC X(27)  VALUE "(1-ADD, 2-CHANGE, 3-ERASE, ".
       01  MENULINE3 PIC X(17)  VALUE "4-DISPLAY, 5-END)".
       01  MENULINE4 PIC X(23)  VALUE "&REQUIRED SERIAL NO.: ".
       01  RECFRSTNM PIC X(18)  VALUE "ENTER FIRST NAME: ".
       01  RECLASTNM PIC X(17)  VALUE "ENTER LAST NAME: ".
       01  RECFLAG   PIC X      VALUE "I".
           88  REC-FOUND        VALUE "F".
           88  SKIP-REC         VALUE "S".
           88  END-OF-FILE      VALUE SPACES.
       PROCEDURE DIVISION.
           DISPLAY MENULINE1 UPON CONSOLE.
           DISPLAY MENULINE2 MENULINE3 UPON CONSOLE.
           DISPLAY MENULINE4 UPON CONSOLE.
           ACCEPT INPLINE1 FROM CONSOLE.
           IF   FNTYPE < 0 AND FNTYPE > 5  THEN
           OPEN INPUT INFILE OUTPUT OUTFILE
           PERFORM FINDREC UNTIL END-OF-FILE
```

```
           CLOSE INFILE
           CLOSE OUTFILE.
           STOP RUN.
       FINDREC.
           PERFORM READREC.
           IF   EMPSER = SERIALNIN THEN
                MOVE "F" TO RECFLAG
                IF   FNTYPE NOT = 1 THEN
                     IF   FNTYPE = 3 THEN
                          DISPLAY GOODMSG UPON CONSOLE
                          MOVE "S" TO RECFLAG
                          PERFORM COPYREST UNTIL END-OF-FILE
                     ELSE
                          MOVE FRSTNMIN TO FNAME
                          MOVE LASTNMIN TO LNAME
                          PERFORM DISPNAME
                ELSE
                     DISPLAY ERRMSG UPON CONSOLE
                     PERFORM COPYREST UNTIL END-OF-FILE
           ELSE
                IF   NOT END-OF-FILE THEN
                     MOVE EMPRECIN TO EMPRECOUT
                     PERFORM WRITEREC
                ELSE
                     IF   FNTYPE = 1 THEN
                          MOVE SPACES TO FNAME
                          MOVE SPACES TO LNAME
                          PERFORM DISPNAME
                     ELSE
                          DISPLAY ERRMSG UPON CONSOLE.
       DISPNAME.
           DISPLAY FNAME LNAME UPON CONSOLE.
           IF   FNTYPE = 4 THEN
                DISPLAY GOODMSG UPON CONSOLE
                PERFORM COPYREST UNTIL END-OF-FILE
           ELSE
                MOVE EMPSER TO SERIALNOUT
                DISPLAY RECFRSTNM UPON CONSOLE
                ACCEPT  FRSTNMOUT FROM CONSOLE
                DISPLAY RECLASTNM UPON CONSOLE
                ACCEPT  LASTNMOUT FROM CONSOLE
                DISPLAY GOODMSG UPON CONSOLE
                PERFORM WRITEREC
                IF   FNTYPE = 2 THEN
                     MOVE "S" TO RECFLAG
                     PERFORM COPYREST UNTIL END-OF-FILE.
       COPYREST.
           IF   SKIP-REC THEN
                MOVE "F" TO RECFLAG
           ELSE
                IF   NOT END-OF-FILE THEN
                     MOVE EMPRECIN TO EMPRECOUT
                     PERFORM WRITEREC.
           IF   NOT END-OF-FILE THEN
                PERFORM READREC.
       READREC.
           READ INFILE AT END
                MOVE SPACES TO RECFLAG.
       WRITEREC.
           WRITE EMPRECOUT.
```

Example 3: COBOL Application Using a CSL Routine Call

The following VS COBOL II application, called CSLCOB COBOL, calls DMSERP, the CSL routine in VMLIB that accesses the extract/replace function. This particular call is extracting the access mode of the first read/only CMS disk.

CSLCOB COBOL contains the following:

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. SAMPLE.
      *
      * THIS SAMPLE VS COBOL II APPLICATION PROGRAM CALLS
      * EXTRACT/REPLACE VIA DMSCSL TO GET THE ACCESS MODE OF THE FIRST
      * READ/ONLY CMS DISK.
      *
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  RTNNAME  PIC X(8)   VALUE "DMSERP  ".
       01  RETCODE  PIC 99999                 COMP-4.
       01  FUNCT    PIC X(8)   VALUE "EXTRACT ".
       01  NUMARGS  PIC 99999  VALUE 1         COMP-4.
       01  INFONAME PIC X(20)  VALUE "ACCESS_MODE         ".
       01  BUFFER   PIC X(5)   VALUE "     ".
       01  DATATYP  PIC 99999  VALUE 0         COMP-4.
       01  BUFLEN   PIC 99999  VALUE 5         COMP-4.
       01  FLAGS    PIC X(8)   VALUE "00000000".
       01  SRCHTYP  PIC X(4)   VALUE "OR  ".
       01  TOKEN    PIC 99999                 COMP-4.
       01  SARGNAM  PIC X(20)  VALUE "CMS_READ_ONLY_DISK  ".
       01  SARGVAL  PIC X      VALUE "1".
       01  SVALTYP  PIC 99999  VALUE 9         COMP-4.
       01  SVALLEN  PIC 99999  VALUE 1         COMP-4.
       01  SARGTYP  PIC XX     VALUE "EQ".
       PROCEDURE DIVISION.
       CALL-DMSCSL SECTION.
           CALL "DMSCSL" USING RTNNAME, RETCODE, FUNCT, NUMARGS,
      -         INFONAME, BUFFER, DATATYP, BUFLEN, FLAGS,
      -         SRCHTYP, TOKEN, SARGNAM, SARGVAL, SVALTYP,
      -         SVALLEN, SARGTYP.
       RETURN-DMSCSL SECTION.
           DISPLAY "RETCODE  = "  RETCODE  UPON CONSOLE.
           DISPLAY "BUFFER   = "  BUFFER   UPON CONSOLE.
           DISPLAY "DATATYP  = "  DATATYP  UPON CONSOLE.
           DISPLAY "BUFLEN   = "  BUFLEN   UPON CONSOLE.
           STOP RUN.
```

Figure 105. COBOL Application with CSL Routine Call

After executing the above program, here is what would be displayed on your terminal (assuming that the B disk is the first minidisk accessed as read/only):

```
RETCODE  = 00000
BUFFER   = B
DATATYP  = 00032
BUFLEN   = 00001
```

Remember the following notes when coding a VS COBOL II program with a call to a CSL routine:

- Each argument in the parameter list must be called (listed) by name.
- Each variable in the parameter list must be level 01.
- Number variables must be fullwords (at least five but less than ten "9"s) and they must be COMP-4, not zoned decimal.
- Hexadecimal values are displayed as decimal.

Appendix D. FORTRAN Examples

These FORTRAN examples follow:

Example 1: Simple FORTRAN Application

The following is the sample FORTRAN program called WELCOME. This application prompts you to enter your first name and last name. Then it welcomes you to CMS.

WELCOME FORTRAN contains the following

```
      PROGRAM WELCOME
      CHARACTER*8 F,S
      WRITE (6,5)
      READ (5,2) F
      WRITE (6,10)
      READ (5,2) S
      WRITE (6,15) F,S
2     FORMAT (A8)
5     FORMAT (' ENTER YOUR FIRST NAME.')
10    FORMAT (' AND NOW YOUR LAST NAME.')
15    FORMAT (' WELCOME TO CMS, ',A8,1X,A8)
      STOP
      END
```

Figure 106. Simple FORTRAN Application

Example 2: Complete FORTRAN Application

The following FORTRAN program lets the user add, change, delete, or display records in a file of peoples' names, by serial number. Records must be added before they can be changed, deleted or displayed.

This application package consists of two parts:

- The source FORTRAN program, called FORT1 FORTRAN, prompts you for information. Then it adds, changes, deletes, or displays the records of the work file.
- The exec program, called DRIVE2 EXEC. DRIVE2 invokes FORT1. It does the file management for FORT1 using CMS commands. FORT1 creates a temporary work file, so DRIVE2 checks if the file already exists. If the work file exists, DRIVE2 issues an error message and does not call FORT1. If the work file does not exist, DRIVE2 issues the FILEDEF commands and calls FORT1.

  Upon return, DRIVE2 tests the return code set by FORT1. If the return code indicates an incomplete work file, DRIVE2 erases the work file. If the return code indicates a completed work file, DRIVE2 erases the old master file and renames the work file as the new master file. The master file is called NAMES DATA.

To execute this application, issue the following commands:

```
FORTVS2 FORT1
GLOBAL TXTLIB VSF2FORT
GLOBAL LOADLIB VSF2LOAD
LOAD FORT1
GENMOD FORT1
DRIVE2
```

DRIVE2 EXEC contains the following information:

```
&TRACE
STATE WORK DATA A
```

```
&IF &RETCODE GT 0 &GOTO -OK
&TYPE FILE 'WORK DATA A' EXISTS.  ERASE AND TRY AGAIN.
&EXIT
-OK
FILEDEF NAMES DISK NAMES DATA
FILEDEF WORK DISK WORK DATA
FORT1
&IF &RETCODE NE 0 &GOTO -NG
STATE WORK DATA A
&IF &RETCODE GT 0 &GOTO -NF
ERASE NAMES DATA A
RENAME WORK DATA A NAMES DATA A
-NF
&EXIT
-NG
ERASE WORK DATA A
&EXIT
```

FORT1 FORTRAN contains the following information:

```
      IMPLICIT INTEGER (A-Z)
      CHARACTER*6  EMPSER,SERNO
      CHARACTER*16 FNAME,LNAME,BNAME
      CHARACTER*21 MSGOK,MSGNG
      DATA  BNAME /'                '/
      DATA  MSGOK /'1OPERATION COMPLETED.'/
      DATA  MSGNG /'1INCORRECT SERIAL NO.'/
      FOUND = 0
      ENDSW = 0
100   FORMAT ('1ENTER FUNCTION NUMBER ')
200   FORMAT (' (1-ADD, 2-CHANGE, 3-ERASE, 4-DISPLAY, 5-END)')
300   FORMAT (' & REQUIRED SERIAL NO.')
400   FORMAT (I1,1X,A6)
500   FORMAT (A16,A16,A6)
600   FORMAT (A16)
700   FORMAT (' ENTER FIRST NAME:')
800   FORMAT (' ENTER LAST NAME:')
900   FORMAT (' ',A16,1X,A16)
```

```
1000  FORMAT (A21)
      WRITE (6,100)
      WRITE (6,200)
      WRITE (6,300)
      READ (6,400) FNTYPE,EMPSER
      IF (FNTYPE.GT.4) GO TO 70
      OPEN  (UNIT=11, FILE='NAMES')
      OPEN  (UNIT=12, FILE='WORK')
10    READ (11,500,ERR=75,IOSTAT=INT,END=15) SERNO,FNAME,LNAME
      IF (INT.NE.0) GO TO 75
      IF (EMPSER.EQ.SERNO) GO TO 20
      WRITE (12,500,ERR=75,IOSTAT=INT) SERNO,FNAME,LNAME
      GO TO 10
15    FOUND = 0
      ENDSW = 1
      GO TO 25
20    FOUND = 1
25    IF (FNTYPE.EQ.1.AND.FOUND.EQ.0) GO TO 30
      IF (FNTYPE.GT.1.AND.FOUND.EQ.1) GO TO 35
      WRITE (6,1000) MSGNG
      IF (FOUND.EQ.0) GO TO 65
      GO TO 55
30    FNAME = BNAME
      LNAME = BNAME
      GO TO 45
35    IF (FNTYPE.EQ.3) GO TO 40
      GO TO 45
40    WRITE (6,1000) MSGOK
      GO TO 60
45    WRITE (6,900) FNAME,LNAME
      IF (FNTYPE.EQ.4) GO TO 50
      SERNO = EMPSER
      WRITE (6,700)
      READ (5,600) FNAME
      WRITE (6,800)
      READ (5,600) LNAME
50    WRITE (6,1000) MSGOK
55    WRITE (12,500,ERR=75,IOSTAT=INT) SERNO,FNAME,LNAME
      IF (ENDSW.EQ.1) GO TO 65
60    READ (11,500,ERR=75,IOSTAT=INT,END=65) SERNO,FNAME,LNAME
      IF (INT.EQ.0) GO TO 55
65    CLOSE (UNIT=11)
      CLOSE (UNIT=12)
      STOP
70    STOP 10
75    STOP 20
      END
```

Figure 107. Complete FORTRAN Application

Example 3: FORTRAN Application Using a CSL Routine Call

The following VS FORTRAN program, called CSLFORT FORTRAN, contains a call to DMSERP, the CSL routine in VMLIB that accesses the extract/replace function. This particular call is extracting the access mode of the first read/only CMS disk. CSLFORT FORTRAN contains the following:

FORTRAN Application Using CSL Routine Call

```
C
C THIS SAMPLE VS FORTRAN APPLICATION PROGRAM CALLS EXTRACT/REPLACE,
C VIA DMSCSL, TO GET THE ACCESS MODE OF THE FIRST READ/ONLY CMS DISK.
C
      PROGRAM SAMPLE
C
C DMSCSL - EXTERNAL INTERFACE ROUTINE TO EXTRACT/REPLACE
C RNAME  - CSL ROUTINE NAME (EXTRACT/REPLACE)
C RCODE  - RETURN CODE FROM EXTRACT/REPLACE
C FUNCT  - FUNCTION TO BE PERFORMED
C NARGS  - NUMBER OF SEARCH ARGUMENTS
C INAME  - INFORMATION NAME
C BUFFER - VALUE THAT WAS EXTRACTED
C DTYP   - DATA TYPE OF DATA EXTRACTED
C BLEN   - LENGTH OF BUFFER/LENGTH OF EXTRACTED DATA
C FLAGS  - FLAGS
C SRTYP  - LOGICAL TYPE OF SEARCH
C TOKEN  - EXTRACT/REPLACE INTERNAL BOOKKEEPER
C SNAM   - SEARCH ARGUMENT NAME
C SVAL   - SEARCH ARGUMENT VALUE
C VTYP   - SEARCH ARGUMENT VALUE'S DATA TYPE
C VLEN   - SEARCH ARGUMENT VALUE'S LENGTH
C STYP   - COMPARISON TYPE
C
      EXTERNAL      DMSCSL
      CHARACTER*8   RNAME
      INTEGER       RCODE
      CHARACTER*8   FUNCT
      INTEGER       NARGS
      CHARACTER*20  INAME
      CHARACTER*2   BUFFER
      INTEGER       DTYP
      INTEGER       BLEN
      CHARACTER*8   FLAGS
      CHARACTER*4   SRTYP
      INTEGER       TOKEN
      CHARACTER*20  SNAM
      CHARACTER     SVAL
      INTEGER       VTYP
      INTEGER       VLEN
      CHARACTER*2   STYPC
      RNAME  = 'DMSERP'
      FUNCT  = 'EXTRACT'
      NARGS  = 1
      INAME  = 'ACCESS_MODE'
      BLEN   = 2
      FLAGS  = '00000000'
      SRTYP  = 'OR  '
      SNAM   = 'CMS_READ_ONLY_DISK'
      SVAL   = '1'
      VTYP   = 9
      VLEN   = 1
      STYP   = 'EQ'
C
C CALL EXTRACT/REPLACE VIA DMSCSL
C
      CALL DMSCSL (RNAME, RCODE, FUNCT, NARGS, INAME, BUFFER,
     *             DTYP, BLEN, FLAGS, SRTYP, TOKEN, SNAM, SVAL,
     *             VTYP, VLEN, STYP)
C
C DISPLAY RESULTS
C
      WRITE (6,30) ' RCODE  = ', RCODE
      WRITE (6,40) ' BUFFER = ', BUFFER
      WRITE (6,30) ' DTYP   = ', DTYP
      WRITE (6,30) ' BLEN   = ', BLEN
C
  30  FORMAT (A9, I4)
  40  FORMAT (A10, A2)
C
      END
```

After executing the above program, here is what would be displayed on your terminal (assuming that the B disk is the first minidisk accessed as read/only):

```
RCODE  =   0
BUFFER = B
```

```
DTYP   =   1
BLEN   =   1
```

Please keep the following notes in mind when coding a VS FORTRAN program with a call to a CSL routine:

- Use the following statement to declare DMSCSL:

  ```
  EXTERNAL DMSCSL
  ```

- You cannot pass hexadecimal values as literal constants in search arguments in a parameter list (Z80000000 or Z00000000) because the VS FORTRAN compiler treats them as variables names and flags them as too long.
- Hexadecimal constants may be used only as data initialization values.
- To initialize hex values to variables declared as integers, use the following data initialization statement:

  ```
  DATA var1/Z80000000/, var2/Z00000000/
  ```

Appendix E. PL/I Example

The following OS PL/I Version 2 program, called CSLPLI PLI, calls DMSERP, the CSL routine in VMLIB that accesses the Extract/Replace function. This particular call is extracting the access mode of the first read/only CMS disk.

CSLPLI PLI contains the following:

```
SAMPLE:  PROCEDURE OPTIONS(MAIN);

/* This sample PL/I application program calls Extract/Replace, */
/* via DMSCSL, to obtain the access mode of the first          */
/* read/only CMS disk.                                         */

/* DMSCSL   - external interface routine to Extract/Replace    */
/* RTNNAME  - CSL routine (Extract/Replace)                    */
/* RETCODE  - return code from Extract/Replace                 */
/* FUNCT    - function to be performed                         */
/* NUMARGS  - number of search arguments                       */
/* INFONAME - information name                                 */
/* BUFFER   - value that was extracted                         */
/* DATATYP  - data type of data extracted                      */
/* BUFLEN   - length of buffer/length of extracted data        */
/* FLAGS    - flags                                            */
/* SRCHTYP  - logical type of search                           */
/* TOKEN    - Extract/Replace internal bookkeeper              */
/* SARGNAM  - search argument name                             */
/* SARGVAL  - search argument value                            */
/* SVALTYP  - search argument value data type                  */
/* SVALLEN  - search argument value length                     */
/* SARGTYP  - comparison type                                  */

DCL DMSCSL     OPTIONS (ASM INTER) ENTRY,
    RTNNAME    CHAR(8)               INIT('DMSERP  '),
    RETCODE    FIXED BINARY(31),
    FUNCT      CHAR(8)               INIT('EXTRACT '),
    NUMARGS    FIXED BINARY(31)      INIT(1),
    INFONAME   CHAR(20)              INIT('ACCESS_MODE         '),
    BUFFER     CHAR(1)               INIT(' '),
    DATATYP    FIXED BINARY(31),
    BUFLEN     FIXED BINARY(31)      INIT(1),
    FLAGS      CHAR(8)               INIT('00000000'),
    SRCHTYP    CHAR(4)               INIT('OR  '),
    TOKEN      FIXED BINARY(31),
    SARGNAM    CHAR(20)              INIT('CMS_READ_ONLY_DISK  '),
    SARGVAL    CHAR(1)               INIT('1'),
    SVALTYP    FIXED BINARY(31)      INIT(9),
    SVALLEN    FIXED BINARY(31)      INIT(1),
    SARGTYP    CHAR(2)               INIT('EQ');
```

Figure 108. PL/I Program Part 1 of 2

```
/* Call Extract/Replace via CSL                                */

CALL DMSCSL(RTNNAME, RETCODE, FUNCT, NUMARGS, INFONAME,
            BUFFER, DATATYP, BUFLEN, FLAGS, SRCHTYP, TOKEN,
            SARGNAM, SARGVAL, SVALTYP, SVALLEN, SARGTYP);

/* Display results                                             */

PUT EDIT ('RETCODE = ', RETCODE)(A);
PUT SKIP EDIT ('BUFFER  = ', BUFFER)(A);
PUT SKIP EDIT ('DATATYP = ', DATATYP)(A);
PUT SKIP EDIT ('BUFLEN  = ', BUFLEN)(A);

END SAMPLE;
```

Figure 109. PL/I Program Part 2 of 2

After executing the above program, here is what would be displayed on your terminal (assuming that the B disk is the first minidisk accessed as read/only):

```
RETCODE =              0
BUFFER  = B
DATATYP =             32
BUFLEN  =              1
```

Please keep the following notes in mind when coding a PL/I program with a call to a CSL routine:

- Numbers in the parameter list must be declared, initialized, and passed as variables.
- Use the following statement to declare DMSCSL:

```
DCL DMSCSL OPTIONS (ASM INTER) ENTRY;
```

Appendix F. REXX Examples

These REXX examples follow:

Example 1: REXX Application Using the CSL Extract/Replace Routine

The following REXX application, called CSLREXX EXEC, calls DMSERP, the CSL routine in VMLIB that accesses the extract/replace function. This particular call is extracting the access mode of the first read/only CMS disk.

```
/*  CSLREXX EXEC  */

address command
funct='RESET'
retcode=0
call csl 'DMSERP retcode funct'   /* Reset EXTRACT/REPLACE */

/* Extract the access mode of the first read-only disk */

/* routine=DMSERP */
retcode=0
funct='EXTRACT'
numargs=1
infoname='ACCESS_MODE'
buffer=''
datatyp=0
buflen=4
flags='00000000'
srchtyp='OR'
token=0
sargnum='CMS_READ_ONLY_DISK'
sargval='1'
svaltyp=9
svallen=1
sargtyp='EQ'

call csl 'DMSERP retcode funct numargs infoname buffer datatyp',
         'buflen flags srchtyp token',
         'sargnum sargval svaltyp svallen sargtyp'

buffer=strip(buffer)

say 'RETCODE = 'retcode
say 'BUFFER  = 'buffer
say 'DATATYP = 'datatyp
say 'BUFLEN  = 'buflen
exit
```

Example 2: REXX Application Using Namedefs

The following example, called READWRIT EXEC, opens, reads, writes, and closes files using the CSL routines DMSOPEN, DMSREAD, DMSWRITE, and DMSCLOSE. One record is read from one file and then written to another file. This example also uses namedefs to identify the files being used. The namedef commands are in a separate file called OPENSETU EXEC. To run this sample program, the NAMEDEFs in OPENSETU EXEC must be changed to directories and files that the current user has access to. In addition, the input file (INFILE) must point to a non-empty file, and the output file (OUTFILE) must not exist in the target directory.

OPENSETU EXEC contains the following:

```
/*  OPENSETU EXEC  */
'CREATE NAMEDEF GIVEFILE EXEC INFILE'
'CREATE NAMEDEF SERVER8:FAIRLIEA. INDIR'
'CREATE NAMEDEF GETFILE EXEC OUTFILE'
'CREATE NAMEDEF SERVER8:FAIRLIEA. OUTDIR'
```

Figure 110. REXX Application Using Namedefs

READWRIT EXEC contains the following:

```
/*  READWRITE EXEC uses:                                          */
/*    DMSOPEN, DMSREAD, DMSWRITE, DMSCLOSE, CACHE, V,             */
/*    and namedefs.                                               */

EXEC OPENSETU

/*  Setting up variables for the OPEN before the READ.            */
retcode=0
reascode=0
fileid1 = infile indir
fileid1len = length(fileid1)
opentype.1 = 'READ'
opentype.2 = 'CACHE'
opentype.3 = 'V'
opentype = opentype.1 opentype.2 opentype.3
opentypelen = length(opentype)

call csl 'DMSOPEN retcode reascode fileid1 fileid1len opentype',
         'opentypelen tokenr'
say 'DMSOPEN:'
say 'return code is 'retcode
say 'reason code is 'reascode
/*  Setting up variables for the READ.                            */
records=1
datalen=130
buffer=0
bufferlen=130
bytesread=16

call csl 'DMSREAD retcode reascode tokenr records datalen buffer',
         'bufferlen bytesread'
say ' '
say 'DMSREAD:'
say 'return code is 'retcode
say 'reason code is 'reascode

/*  Setting up variables (to commit) for the CLOSE of the         */
/*  input file.                                                   */
commit='COMMIT'
comlen=length(commit)

call csl 'DMSCLOSE retcode reascode tokenr commit comlen'

say 'DMSCLOSE (for the READ):'
say 'return code is 'retcode
say 'reason code is 'reascode

/*  Setting up variables for the OPEN before the WRITE.           */
fileid2 = outfile outdir
fileid2len = length(fileid2)
opentype.1 = 'NEW'
opentype.2 = 'CACHE'
opentype.3 = 'V'
opentype = opentype.1 opentype.2 opentype.3
opentypelen = length(opentype)

call csl 'DMSOPEN retcode reascode fileid2 fileid2len opentype',
         'opentypelen tokenw'

say 'DMSOPEN:'
say 'return code is 'retcode
say 'reason code is 'reascode

call csl 'DMSWRITE retcode reascode tokenw records datalen buffer',
         'bufferlen'
say ' '
say 'DMSWRITE:'
say 'return code is 'retcode
say 'reason code is 'reascode
```

```
/*  Setting up variables (to commit) for the CLOSE of               */
/*  the output file.                                                */
commit='COMMIT'
comlen=length(commit)

call csl 'DMSCLOSE retcode reascode tokenw commit comlen'

say 'DMSCLOSE (for the WRITE):'
say 'return code is 'retcode
say 'reason code is 'reascode

exit
```

Appendix G. VS Pascal Example

The following VS Pascal program, called CSLPASC PASCAL, calls DMSERP, the CSL routine in VMLIB that accesses the extract/replace function. This particular call (1) extracts the access mode of the first read/only CMS disk and then (2) gets the access mode of the next CMS minidisk that is both read/only and is an extension of the S-disk.

CSLPASC PASCAL contains the following:

```
PROGRAM CSLPASC(OUTPUT);

(* This sample VS Pascal application program calls             *)
(* Extract/Replace, via DMSCSL, to obtain the access mode of    *)
(* the first read/only CMS disk and of the next CMS minidisk    *)
(* that is both read/only and an extension of the S-disk.       *)

TYPE
     fstring2  = packed array(.1..2.) of char;
     fstring4  = packed array(.1..4.) of char;
     fstring8  = packed array(.1..8.) of char;
     fstring20 = packed array(.1..20.) of char;

(* PROCA calls Extract/Replace via DMSCSL with one search      *)
(* argument (15 parameters).                                   *)

PROCEDURE PROCA (const P0: fstring8;  var P1: integer;
                 const P2: fstring8;  const P3: integer;
                 const P4: fstring20; var P5: char;
                 var P6: integer;     var P7: integer;
                 const P8: fstring8;  const P9: fstring4;
                 var P10: integer;    const P11: fstring20;
                 const P12: char;     const P13: integer;
                 const P14: integer;  const P15: fstring2);

   PROCEDURE DMSCSL (const P0: fstring8;  var P1: integer;
                     const P2: fstring8;  const P3: integer;
                     const P4: fstring20; var P5: char;
                     var P6: integer;     var P7: integer;
                     const P8: fstring8;  const P9: fstring4;
                     var P10: integer;    const P11: fstring20;
                     const P12: char;     const P13: integer;
                     const P14: integer;  const P15: fstring2);
              FORTRAN;
   BEGIN;
     DMSCSL (P0, P1, P2, P3, P4, P5, P6, P7, P8,
             P9, P10, P11, P12, P13, P14, P15);
   END;
 (* PROCB calls Extract/Replace via DMSCSL with two search      *)
(* arguments (20 parameters).                                  *)

PROCEDURE PROCB (const P0: fstring8;   var P1: integer;
                 const P2: fstring8;   const P3: integer;
                 const P4: fstring20;  var P5: char;
                 var P6: integer;      var P7: integer;
                 const P8: fstring8;   const P9: fstring4;
                 var P10: integer;     const P11: fstring20;
                 const P12: char;      const P13: integer;
                 const P14: integer;   const P15: fstring2;
                 const P16: fstring20; const P17: char;
                 const P18: integer;   const P19: integer;
                 const P20: fstring2);

   PROCEDURE DMSCSL (const P0: fstring8;   var P1: integer;
                     const P2: fstring8;   const P3: integer;
                     const P4: fstring20;  var P5: char;
                     var P6: integer;      var P7: integer;
                     const P8: fstring8;   const P9: fstring4;
                     var P10: integer;     const P11: fstring20;
                     const P12: char;      const P13: integer;
                     const P14: integer;   const P15: fstring2;
                     const P16: fstring20; const P17: char;
                     const P18: integer;   const P19: integer;
                     const P20: fstring2);
              FORTRAN;
   BEGIN;
```

```
     DMSCSL (P0, P1, P2, P3, P4, P5, P6, P7,
             P8, P9, P10, P11, P12, P13, P14,
             P15, P16, P17, P18, P19, P20);
   END;

VAR
    RTNNAME:   FSTRING8;  (* CSL routine being called          *)
    RETCODE:   INTEGER;   (* return code from Extract/Replace  *)
    FUNCT:     FSTRING8;  (* Extract/Replace function          *)
    NUMARGS:   INTEGER;   (* number of search arguments        *)
    INFONAME:  FSTRING20; (* information name                  *)
    BUFFER:    CHAR;      (* contains value that was extracted *)
    DATATYP:   INTEGER;   (* data type of data extracted       *)
    BUFLEN:    INTEGER;   (* buffer length/length of ext. data *)
    FLAGS:     FSTRING8;  (* flags                             *)
    SRCHTYP:   FSTRING4;  (* logical type of search            *)
    TOKEN:     INTEGER;   (* internal bookkeeper               *)
    SARGNAM1:  FSTRING20; (* first search argument name        *)
    SARGVAL1:  CHAR;      (* first search argument value       *)
    SARGNAM2:  FSTRING20; (* second search argument name       *)
    SARGVAL2:  CHAR;      (* second search argument value      *)
    SVALTYP1:  INTEGER;   (* search argument value data type   *)
    SVALTYP2:  INTEGER;   (* search argument value data type   *)
    SVALLEN1:  INTEGER;   (* search argument value data length *)
    SARGTYP:   FSTRING2;  (* comparison type                   *)

BEGIN

    RTNNAME  := 'DMSERP  ';
    FUNCT    := 'EXTRACT ';
    NUMARGS  := 1;
    INFONAME := 'ACCESS_MODE';
    BUFFER   := ' ';
    BUFLEN   := 1;
    SRCHTYP  := 'AND';
    FLAGS    := '00000000';
    SARGNAM1 := 'CMS_READ_ONLY_DISK';
    SARGVAL1 := '1';
    SARGNAM2 := 'ACCESS_MODE_EXTEND';
    SARGVAL2 := 'S';
    SVALTYP1 := 9;
    SVALTYP2 := 32;
    SVALLEN1 := 1;
    SARGTYP  := 'EQ';

    PROCA (RTNNAME, RETCODE, FUNCT, NUMARGS,
           INFONAME, BUFFER, DATATYP, BUFLEN,
           FLAGS, SRCHTYP, TOKEN, SARGNAM1,
           SARGVAL1, SVALTYP1, SVALLEN1, SARGTYP);

(* Display results from first call to Extract/Replace         *)

    WRITELN ('RETCODE = ', RETCODE);
    WRITELN ('BUFFER  = ', BUFFER);
    WRITELN ('DATATYP = ', DATATYP);
    WRITELN ('BUFLEN  = ', BUFLEN);

    NUMARGS := 2;
    FLAGS    := '11000000';
    PROCB (RTNNAME, RETCODE, FUNCT, NUMARGS,
           INFONAME, BUFFER, DATATYP, BUFLEN,
           FLAGS, SRCHTYP, TOKEN, SARGNAM1,
           SARGVAL1, SVALTYP1, SVALLEN1, SARGTYP,
           SARGNAM2, 'S', SVALTYP2, SVALLEN1, SARGTYP);

(* Display results from second call to Extract/Replace        *)

    WRITELN;
    WRITELN ('RETCODE = ', RETCODE);
    WRITELN ('BUFFER  = ', BUFFER);
    WRITELN ('DATATYP = ', DATATYP);
    WRITELN ('BUFLEN  = ', BUFLEN);

END.
```

After executing the above program, here is what would be displayed on your terminal (assuming that the B disk is the first minidisk accessed as read/only, and the Y disk is the first read/only minidisk accessed after the B-disk and as an extension of the S-disk):

```
   RETCODE  =          0
   BUFFER   = B
   DATATYP  =         32
   BUFLEN   =          1

   RETCODE  =          0
   BUFFER   = Y
   DATATYP  =         32
   BUFLEN   =          1
```

Keep the following notes in mind when coding a VS Pascal program with a call to a CSL routine:

- Declare DMSCSL as a FORTRAN routine.
- Parameters should be passed as variables by reference, rather than passing them as literals and constants.
- If you call DMSCSL several times with parameter lists that are defined differently or have different numbers of parameters, declare internal procedures to call different formats of DMSCSL.
- Hexadecimal values are displayed as character or integer, depending on how the variable is declared. To display the values as hexadecimal, use an integer parameter and call the function ITOHS with the integer.
- You cannot use an undeclared variable in a call to DMSCSL.

Appendix H. CPI Communications Examples

These CPI Communications examples follow:

Example 1: CPI Communications User Program in z/VM

The following program, CPICRQST EXEC, is a sample REXX user program that requests a file, receives it, and then displays it. This program works in conjunction with the "Example 2: CPI Communications Resource Manager Program in z/VM" on page 561.

See the *z/VM: REXX/VM Reference* for more information about the REXX statements shown in this program.

```
/*************************  CPICRQST  ******************************/
/*                                                                 */
/*  Description:  This program sends a request for a file to the   */
/*    specified VM Resource Manager (as created by the CPICSERV    */
/*    EXEC).  The contents of the requested file is displayed on   */
/*    the console.                                                 */
/*                                                                 */
/*  External References:                                           */
/*    SAA Common Programming Interface Communications Reference    */
/*    z/VM Connectivity Planning, Administration, and Operation    */
/*    CMREXX COPY:  Contains all of the SAA and VM-specific        */
/*     constants.                                                  */
/*                                                                 */
/*  Detailed Information:                                          */
/*                                                                 */
/*    This is a sample CPI Communications program.  It sends a     */
/*    request for a particular file to the specified resource      */
/*    (server), then receives and displays the data received from  */
/*    the that server.                                             */
/*                                                                 */
/*    For demonstration purposes, there is the ability for the     */
/*    user to request 'n' records, which will be created dynamically */
/*    by the server.                                               */
/*                                                                 */
/*    This program uses some VM-specific CPI Communications        */
/*    routines which may not be portable to other operating systems. */
/*::                                                              */
/* Syntax:     (defaults indicated by asterisk)                    */
/*                                                                 */
/*    CPICRQST fn ft <FROM> resource_id <VIA> local_LU remote_LU   */
/*                                      ( TRACE | NOTRACE*         */
/*                                       SECURITY(PGM | SAME* | NONE) */
/*                                       USERID(userid)            */
/*                                       PASSWORD(password)        */
/*                                       BASIC | MAPPED*           */
/*                                       VERIFY| NOVERIFY*         */
/*                                       FORCERR nnn | NOFORCERR*  */
/*                                       CONFIRM | NOCONFIRM*      */
/*                                       STATS(userid)             */
/*                                       TYPE* | NOTYPE            */
/*::                                                              */
/* Parameters:                                                     */
/*                                                                 */
/*   fn ft     is the filename and filetype of the file to be sent. */
/*             If fn is a number and ft is 'RECORDS', it is treated */
/*             as a request for 'n' dynamically created records.   */
/*                                                                 */
/*   resource_id   is the name of the resource.  It is a           */
/*             CPI-Communications symbolic destination name.  If it */
/*             is the nickname of an entry in an active            */
/*             communications directory (COMDIR) file, it will be  */
/*             resolved accordingly.                               */
/*                                                                 */
/*   SECURITY()  specifies the security level of the conversation. */
```

```
/*             If not specified, the value from the communications    */
/*             directory is used.  If there is no communications      */
/*             directory entry, the default will be NONE.             */
/*                                                                    */
/*   USERID()  is the userid to be sent to the remote LU for          */
/*             verification.  Only valid with SECURITY(PGM).          */
/*                                                                    */
/*   PASSWORD()  is the password to be sent to the remote LU for      */
/*             verification.  Only valid with SECURITY(PGM).          */
/*                                                                    */
/*   BASIC     Indicates the conversation type.  The default is       */
/*   MAPPED    MAPPED.                                                */
/*                                                                    */
/*   TRACE     specifies whether certain CPI-C routines are traced.   */
/*   NOTRACE   The default is NOTRACE.                                */
/*                                                                    */
/*   VERIFY    specifies whether the data returned by the server      */
/*   NOVERIFY  is to be verified.  If VERIFY is specified, the        */
/*             file must be available on an accessed disk or          */
/*             directory.  Also, the VERIFY option will suppress      */
/*             the display of the returned data.  The default is      */
/*             NOVERIFY.                                              */
/*                                                                    */
/*   FORCERR nnn   specifies that a SEND_ERROR is to be issued after  */
/*   NOFORCERR  the indicated number of records are received.  If nnn */
/*              is larger than the number of records, then the        */
/*              SEND_ERROR will be issued immediately before the      */
/*              conversation is deallocated.  NOTE: Log data is only  */
/*              available when the BASIC option is specified.         */
/*              The default is NOVERIFY.                              */
/*                                                                    */
/*   TYPE       specifies whether the received records are displayed. */
/*   NOTYPE     at the console.  The default is TYPE.                 */
/*                                                                    */
/*   STATS()    specifies the user who is to receive statistics data. */
/*              The data is of the form 'N=nnnn T=tttt R=rrrr',       */
/*              where:  nnnn is the number of records                 */
/*                      tttt is the time it took to get the records   */
/*                      rrrr is the number of records per second.     */
/*                                                                    */
/* Return Codes:                                                      */
/*     0  - Everything completed OK                                    */
/* 1-200  - CPI Communications error.  Return code is architected.    */
/*   999  - Error in command syntax                                   */
/*  1xxx  - Error xxx from CMS command                                */
/*  2000  - Runtime syntax error.  Invalid environment or programming */
/*          error has occurred.                                       */
/*  3000  - VERIFY was specified and a record has been received that  */
/*          does not match the expected value.                        */
/*                                                                    */
/**********************************************************************/
Trace Off

arg filename filetype rest '(' options
If filename = '?' then Signal Help

'IDENTIFY (LIFO'
pull . . nodeid .
exitrc = 0
If word(rest,1) = 'FROM' then          /* 'FROM' is optional          */
  parse var rest . resource_id rest
Else
  parse var rest resource_id rest

LU_Name = ''
If word(rest,1) = 'VIA' then           /* 'VIA' is optional          */
  parse var rest . LU_Name
Else
  parse var rest LU_Name
LU_Name = strip(LU_Name)
Call Process_Options
'VMFCLEAR'
                                       /* If verification is required,*/
If verify then                         /* get comparison data.        */
  If datatype(filename,'W') & filetype = 'RECORDS' then
    Do
      file.0 = filename                /* Either generate our own...  */
      Do i = 1 to file.0
        FILE.i = left('TEST RECORD NUMBER' i, 80)
      End
    End
  Else                                 /* ...or read the data file.   */
```

```
    Do
      'ESTATE' filename filetype '*'
      If rc <> 0 then
        Call ErrorExit '001 Error' rc 'from ESTATE', 1000+rc

      'EXECIO * DISKR' filename filetype '* (FINIS STEM FILE.'
      If rc <> 0 then
        Call ErrorExit '002 Error' rc 'from EXECIO.  Unable to read',
                       filename filetype'.', 1000+rc
    End


/* Get CPI Communications constants */
'EXECIO * DISKR CMREXX COPY * (FINIS STEM CPICCONST.'
If rc <> 0 then
  Call ErrorExit '002 Error' rc 'from EXECIO.  Unable to read',
                     'CMREXX COPY.', 1000+rc
Do i = 1 to cpicconst.0; interpret cpicconst.i; end
Address CPIComm                         /* Switch environments         */
Call On Error                           /* Catch CSL errors            */
                                        /* Initialize conversation     */
'CMINIT conversation_id' resource_id 'CM_RC'
If CM_RC <> CM_OK then
  Call EMSG 'Unable to init conversation to resource' resource_id

Select                                  /* Determine security level    */
  When security = 'NONE' then security_level = XC_SECURITY_NONE
  When security = 'SAME' then security_level = XC_SECURITY_SAME
  When security = 'PGM'  then security_level = XC_SECURITY_PROGRAM
  Otherwise                   security_level = ''
End

If security_level <> '' then           /* Not using default security  */
  Do
    'XCSCST conversation_id security_level CM_RC'
    If CM_RC <> CM_OK then
      Call EMSG 'Unable to set security level for conversation',
                conversation_id
  End

If security_userid <> '' then          /* Not using default userid    */
  Do
    id_length = length(security_userid)
    'XCSCSU conversation_id security_userid id_length CM_RC'
    If CM_RC <> CM_OK then
      Call EMSG 'Unable to set security userid for conversation',
                conversation_id
  End

If security_password <> '' then        /* Not using default pw        */
  Do
    id_length = length(security_password)
    'XCSCSP conversation_id security_password id_length CM_RC'
    If CM_RC <> CM_OK then
      Call EMSG 'Unable to set security password for conversation',
                conversation_id
  End

/* If remote LU name is character '0', change it to 8 X'00's     */
Parse var LU_Name local_LU remote_LU
if remote_LU = '0' then
  remote_LU = d2c(0,8)
LU_Name = local_LU remote_LU

Select
  when Local_LU = '' then nop          /* Global/local resource       */
  when Local_LU = '*USERID' then       /* Private resource            */
    do
      plul = length(LU_Name)
      'CMSPLN conversation_id LU_Name plul CM_RC'
    end
  Otherwise                            /* Remote resource via AVS or  */
    plul = length(LU Name)             /* ISFC.                       */
    'CMSPLN conversation_id LU_Name plul CM_RC'
    if modename_parm <> '' then        /*   Modename not required for */
      do                               /*   connections via ISFC.     */
        modenamel = length(modename_parm)
        'CMSMN conversation_id modename_parm modenamel CM_RC'
      end
End

/*********************************************************/
```

```
/*    Display everything we can find out about partner   */
/*********************************************************/
'CMEPLN conversation_id plu plu_length CM_RC'

If CM_RC <> 0 then
  Say 'Extract PLU name failed, rc='CM_RC

plu = left(plu,plu_length)
parse var plu plu1 plu2

tpn_length = 3
'XCETPN conversation_id tpn tpn_length CM_RC'
tpn = left(tpn, tpn_length)

sayrqst = 'Requesting' filename filetype 'from'

Select
  When plu_length = 0 then
    Say sayrqst 'Local or Global resource' tpn
  When plu1 = '*USERID' then
    Say sayrqst 'Private resource' tpn 'owned by user' plu2
  Otherwise
    if plu2 = d2c(0,8) then
      Say sayrqst 'Global resource' tpn 'via ISFC gateway' plu1
    else
      Say sayrqst tpn 'at LU' plu2 'via gateway' plu1
    'CMEMN conversation_id modename modename_length CM_RC'
    If CM_RC <> 0 then
      Say 'Extract Mode name failed'
    else
      if modename_length > 0 then
        Say '  Mode name is' left(modename,modename_length)
End

If convtype = 'BASIC' then             /* Set the conversation type   */
  Do
    'CMSCT conversation_id CM_BASIC_CONVERSATION CM_RC'
    If CM_RC <> CM_OK then
      Call EMSG 'Unable to set conversation type to BASIC'
  End

if confirm then
  'CMSSL conversation_id CM_CONFIRM CM_RC'

/*********************************************************/
/*        Allocate a conversation to the server          */
/*********************************************************/
runtime = time('R')
'CMALLC conversation_id CM_RC'
Call TraceSAA 'CMALLC conversation_id CM_RC'

If CM_RC <> CM_OK then
  Call EMSG 'Unable to allocate conversation to' resource_id

send_type = CM_SEND_AND_PREP_TO_RECEIVE /* What to do after CMSEND    */
'CMSST conversation_id send_type CM_RC'

If CM_RC <> CM_OK then
  Call EMSG 'Unable to set Send type'

request = 'From' nodeid':' filename filetype '*'
send_length = length(request)

If convtype = 'BASIC' then             /* Prefix LL if necessary      */
  Do
    send_length = send_length + 2
    request = d2c(send_length,2) || request
  End
/*********************************************************/
/*           Send the request to the server              */
/*********************************************************/
'CMSEND conversation_id request send_length rts CM_RC'
Call TraceSAA 'CMSEND send_length CM_RC'

If CM_RC <> CM_OK then
  Call EMSG 'Unable to send request to' resource_id

/*********************************************************************/
/* Now we have to receive the data that the server will return to us. */
/* We first get 12 bytes of header information.  It contains:         */
/*     - Logical record length                                       */
/*     - Return code from server (i.e. from EXECIO)                  */
```

```
/*     - Number of records sent                                    */
/* Each of these values is a 4-byte binary field.                  */
/*                                                                 */
/* We then receive the data one logical record at a time, until CMRCV */
/* returns an indication in STATUS that no more data is available.    */
/* The DATA_RECEIVED variable may indicate that data is complete      */
/* if the server sent the data using several calls to CMSEND.         */
/* However, the server will either sever the conversation or switch   */
/* to receive state at the end of the file transmission, and this     */
/* will be reported in the STATUS field.                           */
/*******************************************************************/

If convtype = 'MAPPED' then
  requested_length = 12
Else
  requested_length = 14               /* Allow for LL               */

'CMRCV conversation_id buffer requested_length data_received',
        'received_length status rts CM_RC'

Call TraceSAA 'CMRCV data_received received_length status CM_RC'

If CM_RC <> CM_OK then
  Call EMSG 'Error receiving data on conversation' conversation_id

If convtype = 'BASIC' then            /* Skip LL                    */
  buffer = substr(buffer,3)

parse var buffer lrecl 5 eiorc 9 sendrec 13 .

lrecl = c2d(lrecl)
Say 'LRECL='lrecl',  EXECIO RC='c2d(eiorc)',  Records='c2d(sendrec)
If convtype = 'MAPPED' then           /* Set length of first CMRCV  */
  requested_length = lrecl
Else                                  /* For BASIC, receive LL first */
  requested_length = lrecl+2

recno = 0
compmsg = '******  Data received successfully  ******'
log_data = ''

/*******************************************************/
/*           Start receiving the data records          */
/*******************************************************/
Do until find(CM_SEND_RECEIVED CM_CONFIRM_SEND_RECEIVED, status) > 0,
       | CM_RC = CM_DEALLOCATED_NORMAL
  'CMRCV conversation_id buffer requested_length data_received',
        'received_length status rts CM_RC'

  Call TraceSAA 'CMRCV data_received received_length recno status CM_RC'

  If CM_RC <> CM_OK & CM_RC <> CM_DEALLOCATED_NORMAL then
    Call EMSG 'Error receiving data on conversation' conversation_id

  if find(CM_CONFIRM_RECEIVED CM_CONFIRM_SEND_RECEIVED, status) > 0 then
    do
      say 'Confirming receipt of record'
      'CMCFMD conversation_id CM_RC'
      Call TraceSAA 'CMCFMD CM_RC'
    end

  If data_received <> CM_NO_DATA_RECEIVED then
    Do                                /* We have some data!         */
      recno = recno + 1               /* Increment record number    */
      inrec = left(buffer, received_length)

      If convtype = 'BASIC' then
        Do
          If requested_length = lrecl+2 then  /* Strip LL from record */
            inrec = substr(inrec,3)

          If data_received = CM_COMPLETE_DATA_RECEIVED then
            requested_length = lrecl+2 /* Need to receive LL, too    */
          Else
            requested_length = lrecl   /* Only data, no LL           */
        End
      If ¬verify then                 /* If we don't have to verify */
        if typing then                /* and we are supposed to     */
          say inrec                   /* display data, do so.       */
        else nop
      Else                            /* Otherwise, validate it.    */
        If ¬Valid_Data(recno, inrec, lrecl) then
```

```
          do                            /* If it's no good, stop now.  */
            compmsg = '******  Conversation Ended  ******'
            exitrc = 3000
            leave
          end

      /*********************************************************/
      /* See if user wants to force an error.  If so, set the  */
      /* log data, issue a SEND_ERROR, and stop processing.    */
      /*********************************************************/
      If forcerr > 0 & forcerr = recno then
        do
          log_data = 'User forced error after record' recno
          log_data_length = length(log_data)
          'CMSLD conversation_id log_data log_data_length CM_RC'
          CALL TRACESAA 'CMSLD CM_RC'
          'CMSERR conversation_id rts CM_RC'
          CALL TRACESAA 'CMSERR CM_RC'
          compmsg = '******' log_data '******'
          leave
        end
    End
End

/* Give user another opportunity to force an error.           */
If forcerr > 0 & log_data = '' then  /* Only allowed if SEND_ERROR   */
  Do                                 /* not already issued.         */
    log_data = 'User forced error after all data received'
    log_data_length = length(log_data)
    'CMSLD conversation_id log_data log_data_length CM_RC'
    'CMSERR conversation_id rts CM_RC'
    CALL TRACESAA 'CMSERR CM_RC'
    compmsg = '******' log_data '******'
  End

Say compmsg                          /* Display completion message  */

'CMDEAL conversation_id CM_RC'       /* Deallocate conversation     */
runtime = time('E')
If CM_RC <> CM_OK then
  Call EMSG 'Error deallocating conversation' conversation_id

stats = 'N='recno 'T='format(runtime,,2) 'R='format(recno/runtime,,2)
if stats_user <> '' then
  address command 'CP MSG' stats_user stats
exit exitrc

  /*-------------------- Subroutines ----------------------*/

TraceSAA:
  /*********************************************************/
  /* Display variables and their values.  The routine name */
  /* and a list of variables is passed.  If the variable   */
  /* name is 'CM_RC', then the pseudonym will be displayed */
  /* instead of its numeric value.  The conversation state */
  /* will also be displayed.                               */
  /*********************************************************/
  If traceit <> 'YES' then return
  Signal OFF Error

  Arg SAAroutine plist
  Say '****' SAAroutine '****'

  'CMECS conversation_id state ECS_RC'
  if ECS_RC = CM_OK then
    Say 'Conversation State =' CM_CONVERSATION_STATE.state

  Do i = 1 to words(plist)
    SAAvar = word(plist,i)
    Select
      when i = words(plist) then  /* LAST VARIABLE IN LIST IS RC */
        say '  'value('CM_RETURN_CODE.'SAAvar)
      when Saavar = 'DATA_RECEIVED' then
        say '  'CM_DATA_RECEIVED.data_received
      When SAAvar = 'STATUS' then
        say '  'CM_STATUS_RECEIVED.status
      otherwise
        Say '  'SAAvar':' value(SAAvar)
    End
  End
  if value(saavar) = CM_PRODUCT_SPECIFIC_ERROR then
    Call PSE_CHECK
```

```
return
Valid_Data:
  /*********************************************************/
  /*  Verify that received record contains what we think   */
  /*  it should contain.  FILE stem was set during         */
  /*  initialization.  If record isn't valid, set the      */
  /*  Log Data and issue a SEND_ERROR.                     */
  /*********************************************************/
  parse arg record#, input_record, record_length
  check_data = left(FILE.record#,record_length)
  If input_record <>  check_data then
    Do
      log_data = 'Received record did not match source record' recno
      log_data_length = length(log_data)
      Say log_data
      Say "Expected: '"strip(check_data)"'"
      Say "Received: '"strip(input_record)"'"
      hexrec = c2x(strip(input_record))
      hexrec1 = ''
      hexrec2 = ''
      do z = 1 to length(hexrec) by 2
        hexrec1 = hexrec1 || substr(hexrec,z,1)
        hexrec2 = hexrec2 || substr(hexrec,z+1,1)
      end
      Say "    (Hex) '"hexrec1"'"
      Say "           "hexrec2
      'CMSLD conversation_id log_data log_data_length CM_RC'
      'CMSERR conversation_id rts CM_RC'
      CALL TRACESAA 'CMSERR CM_RC'
      return 0
    End
  Else
    return 1

Process_Options:
  /*********************************************************/
  /*  Check specified options.                             */
  /*********************************************************/
  syntax_rc = 99

  If resource_id = '' then
    Call ErrorExit '001 No resource name specified', syntax_rc

  If filename = '' then
    Call ErrorExit '002 No filename specified', syntax_rc
  If filetype = '' then
    Call ErrorExit '003 No filetype specified', syntax_rc
                                          /* Set defaults:          */
  convtype = 'MAPPED'                     /* Mapped conversation    */
  security = ''                           /* Don't override security*/
  traceit = 'NO'                          /* No tracing             */
  verify = 0                              /* Don't check data       */
  forcerr = 0
  typing = 1
  confirm = 0
  security_userid = ''
  security_password = ''
  modename_parm = ''
  stats_user = ''

  Do i = 1 to words(options)              /* Scan options           */
    option = word(options, i)
    Select
      When option = 'VERIFY' then verify = 1
      When option = 'NOVERIFY' then verify = 0
      When option = 'TRACE' then traceit = 'YES'
      When option = 'NOTRACE' then traceit = 'NO'
      When option = 'TYPE' then typing = 1
      When option = 'NOTYPE' then typing = 0
      When option = 'MAPPED' then convtype = 'MAPPED'
      When option = 'BASIC' then convtype = 'BASIC'
      When left(option,6) = 'STATS(' then
        parse var option '(' stats_user ')'
      When left(option,7) = 'USERID(' then
        parse var option '(' security_userid ')'
      When left(option,9) = 'PASSWORD(' then
        parse var option '(' security_password ')'
      When left(option,9) = 'MODENAME(' then
        parse var option '(' modename_parm ')'
      When option = 'SECURITY(PGM)' then security = 'PGM'
      When option = 'SECURITY(NONE)' then security = 'NONE'
      When option = 'SECURITY(SAME)' then security = 'SAME'
```

```
      When option = 'CONFIRM' then confirm = 1
      When option = 'NOCONFIRM' then confirm = 0
      When option = 'NOFORCERR' then forcerr = 0
      When option = 'FORCERR' then
        do
          i = i + 1
          forcerr = word(options,i)
          if forcerr = '' then forcerr = 1
        end
      Otherwise Call ErrorExit '004 Invalid option:' option, syntax_rc
    End
  End
Return

ErrorExit:
  /*********************************************************/
  /*  An error was detected by the program.                */
  /*  Display message and exit with specified return code. */
  /*********************************************************/
  parse arg nnn msg, exitrc
  Say 'CPICRQST'nnn msg
  exit exitrc

ERROR:
  /*********************************************************/
  /*  A host command has failed.  CSL errors will be       */
  /*  translated into a more meaningful error message.     */
  /*********************************************************/
  if rc > 0 then
    parmnum = right(rc,3,'0')+0
  rtn = strip(word(sourceline(sigl),1),'B',"'")

  Select
   When rc = -3 then
     do
       say "CPICRQST005 Routine '"rtn"'not available"
       return
     end
   When rc < -28000 then
     emsg = 'Invalid variable name for parameter number' parmnum
   When rc < -27000 then
     emsg = 'Invalid data type for parameter number' parmnum
   When rc < -26000 then
     emsg = 'Incorrect data length for parameter number' parmnum
   Otherwise
     emsg = 'Return code was' rc
  End

  Say
  Say '==> Run-time error encountered in line' sigl
  Say '==>' emsg
  Say '==>' strip(sourceline(sigl))
  exit 2000
EMSG:
  /*********************************************************/
  /*  An execution-time error has occurred.  If it was a   */
  /*  CSL error, then the return code will be translated   */
  /*  into a more meaningful error message.                */
  /*********************************************************/
  parse arg msg
  Say msg
  CM_RC = CM_RC + 0
  If symbol('CM_Return_Code.'CM_RC) = 'VAR' then
    Say 'RC =' CM_Return_Code.CM_RC
  Else
    Say 'RC =' CM_RC

  if CM_RC = CM_PRODUCT_SPECIFIC_ERROR then
    Call PSE_CHECK

  Exit CM_RC

PSE_CHECK:
  address command 'EXECIO * DISKR CPICOMM LOGDATA * (FINIS STEM LOG.    '
  parse value value('log.'log.0) with . rest
  if subword(rest,1,2) = 'CMSIUCV CONNECT' then
    do
      IUCVERR. = 'Unknown error'
      IUCVERR.1011 = 'Resource or gateway not found'
      IUCVERR.1013 = 'You have exceeded your maximum',
                     'number of connections'
      IUCVERR.1014 = 'Partner has exceeded maximum number',
```

```
                     'of connections'
      IUCVERR.1015 = 'Not authorized to connect to resource'
      IUCVERR.1029 = 'Not authorized to act for another user'
      IUCVERR.1040 = 'Invalid locally known LU name'
      IUCVERR.1041 = 'Invalid mode name'
      IUCVERR.1047 = 'Invalid security subfields in FMH-5'
      IUCVERR.1049 = 'Partner does not allow connections using',
                     'SECURITY(NONE)'
      IUCVERR.1052 = 'No APPCPASS statement found in your VM',
                     'directory entry'
      IUCVERR.1053 = 'Invalid TPN length'
      IUCVERR.1054 = 'Invalid TPN'
      IUCVERR.1089 = 'Path does not support SYNCLVL=SYNCPT'
      parse var rest . . . . . cmsiucv .
      say   '  ['IUCVERR.cmsiucv '('cmsiucv')]'
    end
  else
    Say strip(rest)
  return

HELP:
/*Signal Help1; Help1: helpstart = sigl + 1; Signal Help2;  */
/*
HELP2: Signal Help3;
Help3: helpend = sigl - 1
*/
Do i = 1 by 1 until left(sourceline(i),4) = '/*::'; end
Do i = i+1 by 1 while left(sourceline(i),4) <> '/*::'
  parse value sourceline(i) with '/*' text '*/'
  Say text
End
say "'fn ft' can be specified as 'n RECORDS' for demonstration purposes."
say "See the prolog of this exec for more information."
exit 0
```

Example 2: CPI Communications Resource Manager Program in z/VM

The following program, CPICSERV EXEC, is a sample REXX resource manager program that takes a request for a file and then sends the contents of the file back to the requesting program. This program works in conjunction with the “Example 1: CPI Communications User Program in z/VM” on page 553.

See the *z/VM: REXX/VM Reference* for more information about the REXX statements shown in this program.

```
/*************************  CPICSERV  ******************************/
/*                                                                 */
/*  Description:                                                   */
/*    This program brings up a VM Resource Manager and handles     */
/*    requests for files.  The partner program is CPICRQST EXEC.   */
/*                                                                 */
/*  External References:                                           */
/*    SAA Common Programming Interface Communications Reference    */
/*    z/VM Connectivity Planning, Administration, and Operation    */
/*    CMREXX COPY:  Contains all of the SAA and VM-specific        */
/*     constants.                                                  */
/*                                                                 */
/*  Detailed Information:                                          */
/*                                                                 */
/*    This program is a sample of how to create a VM CPI           */
/*    Communications-based resource manager.  In response to a     */
/*    request for a file from the partner program (CPICRQST), this */
/*    program will send the contents of that file back to the      */
/*    requester.                                                   */
/*                                                                 */
/*    This program uses some VM-specific CPI Communications        */
/*    routines which may not be portable to other operating systems.*/
/*                                                                 */
/* Syntax:                                                         */
/*                                                                 */
/*    CPICSERV <resource_id>  <( options >                         */
/*                                                                 */
/*    Options:  (default indicated by '*')                         */
/*       PRIVATE | LOCAL | GLOBAL                                  */
/*       TRACE | NOTRACE*                                          */
/*       MONITOR userid                                            */
```

```
/*       BUFFER* | NOBUFFER                                            */
/*                                                                     */
/*    Where:                                                           */
/*                                                                     */
/*    resource_id is the name of the resource.  The default is your    */
/*                VM userid.  Certain 3-character prefixes will        */
/*                affect the resource type; see below.                 */
/*                                                                     */
/*    PRIVATE     specifies the type of resource.  If the resource_id */
/*    LOCAL       begins with PRV or CNP then the default is PRIVATE, */
/*    GLOBAL                  LCL or CNL         "           LOCAL,    */
/*                            GBL or CNG         "           GLOBAL    */
/*                Otherwise, the default is GLOBAL.                    */
/*                                                                     */
/*    TRACE       indicates whether the CPI-Communications routines    */
/*    NOTRACE     are to be trace.  The default is NOTRACE.            */
/*                                                                     */
/*    BUFFER      indicates whether data is to be sent in 32k (max)    */
/*    NOBUFFER    blocks (BUFFER) or in lrecl size records (NOBUFFER).*/
/*                                                                     */
/*    MONITOR     specifies the user who is to receive statistical     */
/*                data messages.  A message is generated for every     */
/*                file request.  The format is:                        */
/*                                                                     */
/*                   uuuuuuuu  N=nnnnn  T=ttttt  Bps=bbbbbbb           */
/*                                                                     */
/*                uuuuuuuu is the requesting userid                    */
/*                nnnnn    is the number of records sent               */
/*                ttttt    is the time needed to process the request   */
/*                         (in seconds)                                */
/*                bbbbbbb  is an estimate of the number of bytes       */
/*                         transmitted per second.                     */
/*                                                                     */
/*  LOCAL and GLOBAL resources require VM directory authorization      */
/*  via the IUCV *IDENT statement.  See the  z/VM Connectivity,         */
/*  Administration, and Operation book for more information.           */
/*                                                                     */
/*********************************************************************/
Trace Off

/* Get CPI Communications constants */
'EXECIO * DISKR CMREXX COPY * (FINIS STEM CPICCONST.'
If rc ¬= 0 then exit rc
Do i = 1 to cpicconst.0; interpret cpicconst.i; End

Arg resid '(' options                  /* Get resource id & options   */
resid = word(resid userid(), 1)        /* Resid defaults to userid    */

Call Process_Options                   /* Scan options                */
'VMFCLEAR'

Address CPIComm                        /* Switch environments         */
Call On Error                          /* Trap errors                 */
Signal On NoValue

If server_scope ¬= XC_PRIVATE then     /* Issue a nice message        */
  Do
    /*****************************************************************/
    /* Tell CP about this resource using the Identify_Resource_   */
    /* Manager routine, XCIDRM.                                      */
    /*                                                               */
    /* The server has two primary characteristics:                   */
    /*  - It can only serve one client (user) at a time, but it   */
    /*    can service additional clients after the first without  */
    /*    exiting.  Request are queued simply by avoiding calls   */
    /*    to XCWOE while a conversation is active.                   */
    /*                                                               */
    /*  - It will accept conversations from users who allocate    */
    /*    their conversations with SECURITY(NONE).                   */
    /*****************************************************************/

    service_mode = XC_MULTIPLE         /* Service many clients    */
    secnone = XC_ACCEPT_SECURITY_NONE  /* Accept SECURITY(NONE)   */

    'XCIDRM   resid server_scope service_mode secnone CM_RC'
    /* Parameter check is OK (it means we already own resource).  */
    If CM_RC ¬= CM_OK & CM_RC ¬= CM_PROGRAM_PARAMETER_CHECK then
      Do
        Say 'Unable to manage resource' resid':' CM_RETURN_CODE.CM_RC
        Exit CM_RC
      End
```

```
    if monitor <> '' then
      address command 'CP MSG' monitor 'Server is up'

    Say restype 'resource' resid 'is now up and ready for work.'
    Say
    Say "At 'waiting' message, you can enter one of the following:"
    Say '   TERMINATE   to relinquish control of resource.'
    Say '   STOP        to just halt server.  Resource is still owned.'
    Say '   TRACE       to start CPI Communications trace.'
    Say '   NOTRACE     to stop CPI Communications trace.'
    Say '   <other>     is considered to be a CP or CMS command'
    Say
  End

Do Forever
  If server_scope ¬= XC_PRIVATE then  /* Use Wait_On_Event           */
    Do
      Say
      Say 'Waiting for work, TERMINATE, STOP, TRACE, NOTRACE, CP, or',
          'CMS command'
      'XCWOE resid conversation_id event length buffer WOE_rc'
    End
  Else
    event = XC_ALLOCATION_REQUEST     /* Simulate an XCWOE call  */
  Select
    When event = XC_ALLOCATION_REQUEST then
      Do
        address command 'CP SPOOL CONS PURGE'
        address CMS 'VMFCLEAR'
        'CMACCP conversation_id ACCP_rc'
        If ACCP_rc ¬= CM_OK then
          Say 'Unable to accept conversation, rc =' ACCP_rc

        'XCETPN conversation_id tpn tpnl cm_rc'
        tpn = left(tpn,tpnl)
        say restype 'server' tpn 'running...'

        Call Display_Partner_Info

        'CMECT conversation_id conversation_type CM_RC'
        If conversation_type = CM_MAPPED_CONVERSATION then
          Say 'Conversion type is MAPPED'
        Else
          Say 'Conversion type is BASIC'

        numrecs = 0
        etime = time('R')
        Call Process_Request

        if monitor <> '' then
          do
            etime = time('R')
            msg =  partner_id '  N='numrecs ' T='format(etime,,1),
            ' Bps='format(numrecs*lrecl/etime,,0)
            address command 'CP MSG' monitor msg
          end

        If server_scope = XC_PRIVATE then
          exit
      End

    When event = XC_CONSOLE_INPUT then
      Call User_Input left(buffer,length)

     Otherwise
       Say 'Unexpected' XC_EVENT_TYPE.event 'received'
       Signal Terminate_Server
   End
End /* Do forever */
Exit 999                            /* Should never reach this point */
Process_Request:
/*********************************************************************/
/* A request for a file has been received                            */
/*********************************************************************/
confirm = 0
requested_file = ''
length = 50                          /* Max length of initial rcv   */
                                     /* Receive file id             */
Do until CMRCV_RC ¬= CM_OK ,
  & left(CM_RETURN_CODE.CMRCV_RC,16) <> 'CM_PROGRAM_ERROR'
  'CMRCV conversation_id buffer length data_flag',
        'received_length status_flag rts_flag CM_RC'
```

```
  CMRCV_RC = CM_RC
  Call TraceSAA 'CMRCV data_flag received_length status_flag CM_RC'

  Select
    When CMRCV_RC = CM_DEALLOCATED_NORMAL then
      Say 'Conversation with' partner_id 'ended'

    When CMRCV_RC = CM_OK then
      Do
        If data_flag ¬= CM_NO_DATA_RECEIVED then
          Do
            If conversation_type = CM_BASIC_CONVERSATION then
              buffer = substr(buffer, 3, received_length-2)
            Else
              buffer = left(buffer, received_length)
            requested_file = requested_file || buffer
          End

        Select
          when status_flag = CM_SEND_RECEIVED then
            Call Send_It
          when status_flag = CM_CONFIRM_SEND_RECEIVED then
            do
              confirm = 1
              'CMCFMD conversation_id CM_RC'
              say 'User desires confirmation of records sent'
              Call TraceSAA 'CMCFMD CM_RC'
              Call Send_It
            end
          when status_flag = CM_CONFIRM_DEALLOC_RECEIVED then
            do
              'CMCFMD conversation_id CM_RC'
              say 'Confirming deallocation'
              Call TraceSAA 'CMCFMD CM_RC'
              leave
            end
          otherwise nop
        End
      End
    Otherwise
      Call Analyze_CM_RC 'CMRCV'
  End
End
return

Send_It:
/*********************************************************************/
/*  The request has been received.  It is either:                    */
/*                                                                   */
/*     1. The file name and type of a file the user wishes to be     */
/*        transmitted, or                                            */
/*                                                                   */
/*     2. A number indicating how many program-generated sequential  */
/*        records are to be sent.  This is indicated by a numeric    */
/*        file name and a file type of 'RECORDS'                     */
/*                                                                   */
/*  We send the file back to the user in the following format:       */
/*    - A 12-byte header consisting of LRECL, EXECIO return code     */
/*      and number of records sent.                                  */
/*    - The file contents                                            */
/*********************************************************************/
Parse VAR requested_file . . fn ft fm .

If datatype(fn,'W') & ft = 'RECORDS' then
  Do                                  /* User requested that dummy   */
    lrecl = 80                        /* records be generated.       */
    whatsent = fn 'records'
    erc = 0
    file.0 = fn
    senddata = ''
    Do i = 1 to file.0
     senddata = senddata || left('TEST RECORD NUMBER' i, lrecl)
    End
  End
Else                                  /* Look for the requested file */
  Do
    Address Command                   /* Talk to CMS for a minute... */
    'SET CMSTYPE HT'                  /* Find first occurrence of file*/
    'MAKEBUF'                         /* in the search order.        */
    cmscmd = 'LISTFILE'
    'LISTFILE' fn ft fm '(FIFO ALLOC NOHEADER'
    erc = rc
```

```
    If rc = 0 then                    /* If file found, get the      */
      Do                              /* length of longest record.   */
        pull . . . . lrecl .
        cmscmd = 'EXECIO'
        'EXECIO * DISKR' fn ft fm '(FINIS STEM FILE.'
        erc = rc
        If rc = 0 then                /* Compress file into a single */
          Do                          /* physical record composed of */
            senddata = ''             /* logical records of length   */
            Do i = 1 to file.0        /* 'lrecl'.                    */
              senddata = senddata || left(file.i, lrecl)
            End
            whatsent = "file '"fn ft fm"'"
          End
      End

    If erc ¬= 0 then                  /* LISTFILE or EXECIO error    */
      Do
        file.0 = 0                    /* No records to process       */
        senddata = '==> Error during' cmscmd 'processing <=='
        lrecl = length(senddata)
        whatsent = 'error message'
      End

    'DROPBUF'
    'SET CMSTYPE RT'
    Address CPIComm
  End

  /*****************************************************************/
  /* Build and send the header with lrecl, EXECIO return code, and */
  /* number of records.                                            */
  /*****************************************************************/
sendhdr = d2c(lrecl,4) || d2c(erc,4) || d2c(file.0,4)
send_size = length(sendhdr)
numrecs = file.0
Drop File.
If conversation_type = CM_BASIC_CONVERSATION then
  Do                                  /* Add LL to front of header   */
    send_size = send_size + 2
    sendhdr = d2c(send_size, 2) || sendhdr
  End
                                      /* Send header                 */
'CMSEND conversation_id sendhdr send_size rts CM_rc'
Call TraceSAA 'CMSEND send_size CM_RC'

If CM_RC ¬= CM_OK then                /* Check for errors            */
  Do
    Call Analyze_CM_RC 'CMSEND of header'
    Return
  End
if confirm then
  do
    'CMSST conversation_id CM_SEND_AND_CONFIRM CM_RC'
    Call TraceSAA 'CMSST CM_RC'
  end

size = length(senddata)
Say 'Sending' whatsent '('size 'bytes) to' partner_id

    /*************************************************************/
    /* Compute the largest record that can be sent with a single  */
    /* CMSEND.  It will be the largest multiple of the LRECL that */
    /* is less than or equal to 32,765.  (32K - 2 bytes for LL)   */
    /*                                                           */
    /* Note that this is not done if NOBUFFER is specified.      */
    /*************************************************************/
if buffering then
  interval = 32765 % lrecl * lrecl

else
  interval = lrecl

If traceit then
  Say 'Maximum send length =' interval

Do while size > 0
  send_size = min(size,interval)      /* How much data can we send?  */
  senddata2 = left(senddata,send_size) /* Set up send buffer         */
  Say '...as a record of' right(send_size,5) 'bytes'
  If conversation_type = CM_BASIC_CONVERSATION then
    Do                                /* Add LL to front of record   */
```

```
      send_size = send_size + 2
      senddata2 = d2c(send_size, 2) || senddata2
    End
                                      /* Send it to user             */
  'CMSEND conversation_id senddata2 send_size rts CM_rc'
  Call TraceSAA 'CMSEND send_size rts CM_RC'

  If CM_RC ¬= CM_OK then              /* Check for errors            */
    Do
      Call Analyze_CM_RC 'CMSEND'
      Return
    End

  size = size - interval              /* Figure out how much data    */
  If size > 0 then                    /* remains to be sent.         */
    senddata = substr(senddata,interval+1)
end
'CMPTR conversation_id CM_RC'         /* Turn the conversation around */
call TraceSAA 'CMPTR conversation_id CM_RC'

If CM_RC ¬= CM_OK then
  Call Analyze_CM_RC 'CMPTR'

Return                               /* Go back & wait for next event */

Analyze_CM_RC:

/*********************************************************************/
/* Figure out what when wrong the CPI-Communications call.           */
/*                                                                   */
/* If a product-specific error occurs, detailed information about the */
/* error is written to the CPICOMM LOGDATA file.  We will extract    */
/* the data and display it.                                          */
/*                                                                   */
/* If the client (user) issues a SEND_ERROR (indicating that the data */
/* received was corrupted), will assume that Log Data was sent.      */
/* Note that Log Data is only available on BASIC conversations.      */
/* The Log Data is in the form of an SNA Error Log GDS variable.     */
/* See the VM/ESA: CP Programming Services book for the format of the  */
/* Error Log GDS variable.                                           */
/*********************************************************************/
  parse arg whocalled
  Select
    when CM_RC = CM_PRODUCT_SPECIFIC_ERROR then
      Do
        address command 'EXECIO * DISKR CPICOMM LOGDATA *',
                                        '(FINIS STEM LOG.'
        parse value value('log.'log.0) with . rest
        Say strip(rest)
      End
    when index(CM_Return_Code.CM_RC,'PROGRAM_ERROR') > 0 then
      If conversation_type = CM_BASIC_CONVERSATION then
        Do
          address command 'EXECIO * DISKR CPICOMM LOGDATA *',
                                          '(FINIS STEM LOG.'
          Log_Data = value('log.'log.0)
          gds3 = c2d(substr(log_data,5,2))
          user_data = substr(log_data,GDS3+7)
          Say CM_Return_Code.CM_RC':' user_data
        End
      Else
        Say CM_Return_Code.CM_RC '(no additional information available)'

    otherwise
      Say CM_Return_Code.CM_RC 'on' whocalled 'to' partner_id
  End
Return
User_Input:
/*********************************************************************/
/* User typed something.  Figure out what to do.                     */
/*********************************************************************/
  arg user_buffer

  Select
    When user_buffer = '' then nop

    When user_buffer = 'STOP' then
      Do
        Say 'Terminating immediately due to STOP command.'
        exit
      End
```

```
    When user_buffer = 'TERMINATE' then
      Signal Terminate_server

    When user_buffer = 'TRACE' then
      traceit = 1

    When user_buffer = 'NOTRACE' then
      traceit = 0
    Otherwise
      Address CMS user_buffer
  End
Return

Terminate_Server:
/*********************************************************************/
/* Bring down server.  If we are in the middle of a conversation,    */
/* deallocate it.  We will terminate our ownership of the resource.  */
/*********************************************************************/
  'CMECS conversation_id conv_state CMECS_RC'
  If find(CM_SEND_STATE CM_SEND_PENDING_STATE, conv_state) > 0 then
    Do
      'CMDEAL conversation_id CM_RC'
      Call TraceSAA 'CMDEAL CM_RC'
    End

  Say "Terminating resource '"Save_resid"'"
  'XCTRRM  SAVE_RESID TRRM_RC'

  Exit TRRM_RC
Display_Partner_Info:
/*********************************************************************/
/* Display everything we can find out about our communications       */
/* partner.                                                          */
/*********************************************************************/
  'XCECSU conversation_id partner_id partner_id_length CM_RC'
  If partner_id_length = 0 then
    partner_id = 'Someone'
  Else
    partner_id = strip(left(partner_id,partner_id_length))

  'CMEPLN conversation_id plu plu_length cm_rc'
  If cm_rc ¬= 0 then
    Say 'Extract PLU name failed, rc='cm_rc

  plu = left(plu,plu_length)
  If traceit then
    Say 'PLU='plu'  Length='plu_length

  parse var plu plu1 plu2
  If plu1 = '' then plu1 = '*IDENT'

  If find('*IDENT *USERID', plu1) > 0 then
    Say 'Request from' partner_id 'in this TSAF collection'
  Else
    Do
      'CMEMN conversation_id modename modename_length cm_rc'
      If cm_rc ¬= 0 then
        Say 'Extract Mode name failed'

      Say 'Request from' partner_id 'at LU' plu2 'via gateway' plu1'.',
          'Mode name is' left(modename,modename_length)'.'
    End
Return

TraceSAA:
/*********************************************************************/
/* Display the results of a CPI-Communications call.                 */
/*********************************************************************/
  If ¬traceit then return

  Arg SAAroutine plist
  Say '****' SAAroutine '****'

  'CMECS conversation_id conv_state CMECS_RC'
  If CMECS_RC = CM_OK then
    Say 'Conversation state:' CM_CONVERSATION_STATE.conv_state
  else
    say 'Unable to determine conversation state:',
        CM_Return_Code.CMECS_RC

  Do i = 1 to words(plist)
    SAAvar = word(plist,i)
```

```
    If SAAvar = 'CM_RC' then
      Say 'CM_RC:' value('CM_RETURN_CODE.CM_RC')
    Else
      Say SAAvar':' value(SAAvar)
  End
Return

Process_Options:
/*********************************************************************/
/* Check for any options the user specified.  We will determine:     */
/*   - Server scope (Global, Local, or Private)                      */
/*   - Whether to activate tracing                                   */
/*********************************************************************/
  restype = 'Global'                  /* Set defaults               */
  server_scope = XC_GLOBAL
  monitor = ''
  buffering = 1
  traceit = 0
  Select                              /* Check for special prefixes  */
    When find('PRV CNP', left(resid,3)) > 0 then
      options = 'PRIVATE' options

    When resid = 'PRIVATE' then
      options = 'PRIVATE' options

    When find('GLB CNG', left(resid,3)) > 0 then
      options = 'GLOBAL' options
    When find('LCL CNL', left(resid,3)) > 0 then
      options = 'LOCAL' options
    Otherwise nop
  End

  Do i = 1 to words(options)          /* Verify all options          */
    keyword = word(options,i)

    Select
      when keyword = 'MONITOR' then
        do
          monitor = word(options, i+1)
          i = i + 1
        end
      when keyword = 'NOBUFFER' then
        buffering = 0

      when keyword = 'PRIVATE' then
        parse value XC_PRIVATE 'Private' with server_scope restype .

      when keyword = 'LOCAL' then
        parse value XC_LOCAL 'Local' with server_scope restype .

      when keyword = 'GLOBAL' then
        parse value XC_GLOBAL 'Global' with server_scope restype .

      when keyword = 'TRACE' then     /* Activate tracing            */
        traceit = 1

      when keyword = 'NOTRACE' then   /* Deactivate tracing          */
        traceit = 0

      otherwise
        Say 'Unknown option:' keyword
        exit 24
    End
  End
  Save_resid = resid                  /* Remember resource id        */
Return

NOVALUE:
  Say '+++ NOVALUE Condition raised in line' sigl':'
  Say '+++' sourceline(sigl)
  Signal Terminate_Server

ERROR:
  failing_routine = strip(word(sourceline(sigl), 1),'B', "'")
  say '('failing_routine 'routine not available)'
  return
```

Example 3: Synchronizing Multiple Updates Using CRR and CPI Communications

This sample consists of the user REXX application and the target REXX application associated with CRR scenario described in “Scenario 3: Synchronizing Multiple Updates” on page 511.

User Application, CRREXMP1 EXEC

The following is the user REXX application, CRREXMP1 EXEC:

```
/*--------------------------------------------------------------------*/
/*  CRREXMP1 EXEC -- User application                                 */
/*--------------------------------------------------------------------*/

/*--------------------------------------------------------------------*/
/*   Main Logic of CRREXMP1:                                          */
/*                                                                    */
/*       Initialize program constants                                 */
/*       Verify the required SFS directories exist                    */
/*       Get the toy to be added to the list                          */
/*       Set synchronization point options and transaction tag        */
/*       Update the first file                                        */
/*       If first update fails then exit                              */
/*          Else do                                                   */
/*             Update the second file                                 */
/*             If second update failed then exit                      */
/*                Else request third update                           */
/*          End                                                       */
/*--------------------------------------------------------------------*/

call Initialize
call Verify_SFS
call Get_Toy
call Setup_Syncpoint

call Update_File file1, dir1
call Update_File file2, dir2

call Request_Update

CRREXMP1_Exit:

   call Display ' '
   call Display 'CRREXMP1 complete.'
   call Display ' '

   exit

/*--------------------------------------------------------------------*/
/*  INITIALIZE                                                        */
/*                                                                    */
/*  This sub-routine will set up various constants and variables      */
/*  that will be used throughout this EXEC.                           */
/*                                                                    */
/*  NOTES:                                                            */
/*          -  Variables dir1 and dir2 must be set to SFS             */
/*             directories that the userid running this EXEC owns     */
/*             or has write access to.  They cannot be in the same    */
/*             filepool as the directory the CRREXMP2 EXEC uses.      */
/*                                                                    */
/*          -  Files SRRREXX COPY and CMREXX COPY provide constants   */
/*             and literals for SAA resource recovery and CPI         */
/*             Communications.  They must exist on an accessed        */
/*             minidisk or SFS directory.                             */
/*--------------------------------------------------------------------*/

Initialize:

   'vmfclear'

   dash                 = '*' || copies('-', 68) || '*'
   blanks25             = copies(' ', 25)
   conversation_active  = 0
   errors_found         = 0
   commit_turned_backout = 0
```

```
   call Display dash
   x = 'CRREXMP1:  Sample Coordinated Resource Recovery Application'
   call Display x
   call Display dash
   call Display ' '

   file1 = 'CHILDS LIST'
   file2 = 'TOYSTORE ORDERS'
   dir1  = 'SERVER3:HEALDJS.CRRDIR1.'
   dir2  = 'SERVER3:HEALDJS.CRRDIR2.'

   copy_file.1 = 'CMREXX COPY *'
   copy_file.2 = 'SRRREXX COPY *'
   do i = 1 to 2
      address command 'ESTATE' copy_file.i
      if (rc ¬= 0) then call Error copy_file.i 'file not found'
         else do
            'execio * diskr' copy_file.i '(finis stem PSEUDONYM.'
            do j = 1 to pseudonym.0
               interpret pseudonym.j
            end
         end
   end

   return

/*--------------------------------------------------------------------*/
/*  VERIFY_SFS                                                        */
/*                                                                    */
/*  This sub-routine will verify that required SFS directories      */
/*  are available for the EXEC to run.                                */
/*--------------------------------------------------------------------*/

Verify_SFS:

   call Display ' '
   x = 'CRREXMP1:  Verifying required directories are available.'
   call Display x
   call Display ' '

   len1 = length(dir1)
   call csl('DMSEXIDI exidi_rc exidi_rs dir1 len1 NOCOMMIT 8')
   erc = overlay(exidi_rc, '          ')
   x = '           DMSEXIDI   Retcode =' erc  'Reascode =' exidi_rs
   call Display x
   if (exidi_rc ¬= 0) then call Error 'Unexpected result on DMSEXIDI'

   len2 = length(dir2)
   call csl('DMSEXIDI exidi_rc exidi_rs dir2 len2 NOCOMMIT 8')
   erc = overlay(exidi_rc, '          ')
   x = '           DMSEXIDI   Retcode =' erc  'Reascode =' exidi_rs
   call Display x
   if (exidi_rc ¬= 0) then call Error 'Unexpected result on DMSEXIDI'
   call Display ' '

   return
/*--------------------------------------------------------------------*/
/*  GET_TOY:                                                          */
/*                                                                    */
/*  This sub-routine will prompt the user to enter the name of the  */
/*  toy to be added to the list.  A user and time stamp is then     */
/*  added to this, and the resulting string is the record that      */
/*  we will try to add to the CHILDS LIST, TOYSTORE ORDERS, and the */
/*  SANTAS SACK files.  If the user enters a null response, then    */
/*  the value "TRAIN" will be used.                                   */
/*--------------------------------------------------------------------*/

Get_Toy:

   call Display ' '
   x = 'CRREXMP1:  Enter the name of the toy you want'
   call Display x
   pull response
   if (response = '') then response = 'TRAIN'
   response = left(response, 10)
   response = 'Toy Request: ' overlay(response, '          ')
   'id (stack'
   pull id_line
   parse var id_line userid 'AT' node .
   time = left(time(), 5)
   date = date(u)
```

```
   tran = response '==>' node userid date time
   tran_length = length(tran)

   return

/*--------------------------------------------------------------------*/
/*  SETUP_SYNCPOINT:                                                  */
/*                                                                    */
/*  This sub-routine sets some options for synchronization points:  */
/*  First, DMSSSPTO is called to indicate that if resynchronization */
/*  is required, control will be returned to the application if it  */
/*  is not possible to complete the resynchronization immediately.  */
/*                                                                    */
/*  Next, DMSSETAG is called to set the transaction tag for this    */
/*  transaction.  CMS stores this tag on the log to assist an       */
/*  operator in the event that they need to intervene during        */
/*  resynchronization.  In this case, the actual data record that   */
/*  we are attempting to add to the SFS files is used as the tag.   */
/*                                                                    */
/*  An error from either of these routines is displayed, but        */
/*  processing will continue.                                       */
/*--------------------------------------------------------------------*/
Setup_Syncpoint:

   call Display ' '
   call Display ' '
   x = 'CRREXMP1:  Setting Synchronization Point Options'
   call Display x
   call Display ' '
   call csl('DMSSSPTO sspto_rc sspto_rs NOWAIT 6')
   src = overlay(sspto_rc, '          ')
   x = '           DMSSSPTO   Retcode =' src  'Reascode =' sspto_rs
   call Display x
   call Display ' '
   call Display ' '
   x = 'CRREXMP1:  Setting Transaction Tag'
   call Display x
   call Display ' '
   call csl('DMSSETAG setag_rc setag_rs tran tran_length')
   src = overlay(setag_rc, '          ')
   x = '           DMSSETAG   Retcode =' src  'Reascode =' setag_rs
   call Display x
   x = '           Tag =' tran
   call Display x

   return

/*--------------------------------------------------------------------*/
/*  UPDATE_FILE:                                                      */
/*                                                                    */
/*  This sub_routine updates a file in an SFS directory by adding   */
/*  one record.  The file and SFS directory are input to this       */
/*  sub-routine.  If the file is successfully opened and updated,   */
/*  it is then closed with the NOCOMMIT option.  CSL routines       */
/*  issued include:                                                   */
/*                                                                    */
/*  DMSOPEN  ... open the input file in the input directory         */
/*  DMSWRITE ... write the data record to the open file             */
/*  DMSCLOSE ... close the input file in the input directory        */
/*--------------------------------------------------------------------*/

Update_File:

   arg file, dir
   call Display ' '
   call Display ' '
   call Display 'CRREXMP1:  UPDATING <'file'> in <'dir'>'
   call Display ' '

   file_spec = file dir
   len1 = length(file_spec)

   call csl('DMSOPEN open_rc open_rs file_spec len1 WRITE 5 token')
   orc = overlay(open_rc, '          ')
   x = '           DMSOPEN    Retcode =' orc  'Reascode =' open_rs
   call Display x

   if (open_rc = 0) | ((open_rc = 4) & (open_rs = 44030)) then do
      call csl('DMSWRITE write_rc write_rs token 1 tran_length tran 80')
      wrc = overlay(write_rc, '          ')
      x = '           DMSWRITE   Retcode =' wrc 'Reascode =' write_rs
      call Display x
```

```
      if (write_rc = 0) then do
         call csl('DMSCLOSE close_rc close_rs token NOCOMMIT 8')
         crc = overlay(close_rc, '          ')
         x = '           DMSCLOSE   Retcode =' crc  'Reascode =' close_rs
         call Display x
         if (close_rc ¬= 0) then                              ,
            call Error 'Unexpected result on DMSCLOSE'
      end

         else call Error 'Unexpected result on DMSWRITE'
   end

      else call Error 'Unexpected result on DMSOPEN'

   return

/*--------------------------------------------------------------------*/
/*  REQUEST_UPDATE:                                                   */
/*                                                                    */
/*  This sub_routine uses CPI Communications routines to              */
/*  communicate with CRREXMP2, which will update a file and           */
/*  initiate synchronization point processing.  The following         */
/*  routines are used:                                                */
/*                                                                    */
/*  CMINIT ... initialize the protected conversation                  */
/*  CMSSL  ... set sync_level to cm_sync_point (protected)            */
/*  CMALLC ... allocate the protected conversation                    */
/*  CMCFM  ... issue confirm to ensure partner is talking to us       */
/*  CMSEND ... send the transaction data, confirmation request        */
/*  CMPTR  ... request to switch to receive state                     */
/*  CMCFM  ... confirm switch to receive state                        */
/*  CMRCV  ... receive the synchronization point intention            */
/*--------------------------------------------------------------------*/

Request_Update:

   call Display ' '
   call Display ' '
   x = 'CRREXMP1:  Establishing protected conversation to CRREXMP2'
   call Display x
   call Display ' '

   address cpicomm
   'CMINIT conv_1 CRREXMP2 cm_rc'
   x = '   Initialize_Conversation (CMINIT) ..'
   call Display x 'return_code =' cm_return_code.cm_rc
   if (cm_rc ¬= cm_ok) then call Error 'Unexpected return_code on CMINIT'

   conversation_active = 1

   'CMSSL  conv_1 cm_sync_point cm_rc'
   x = '   Set_Sync_Level (CMSSL) ............'
   call Display x 'return_code =' cm_return_code.cm_rc
   if (cm_rc ¬= cm_ok) then call Error 'Unexpected return_code on CMSSL'

   'CMALLC conv_1 cm_rc'
   x = '   Allocate (CMALLC) .................'
   call Display x 'return_code =' cm_return_code.cm_rc
   if (cm_rc ¬= cm_ok) then call Error 'Unexpected return_code on CMALLC'

   'CMCFM  conv_1 rts_received cm_rc'
   x = '   Confirm (CMCFM) ...................'
   call Display x 'return_code =' cm_return_code.cm_rc
   if (cm_rc ¬= cm_ok) then call Error 'Unexpected return_code on CMCFM'
   'CMSEND conv_1 tran tran_length rts_received cm_rc'
   x = '   Send_Data (CMSEND) ................'
   call Display x 'return_code =' cm_return_code.cm_rc
   if (cm_rc ¬= cm_ok) then call Error 'Unexpected return_code on CMSEND'

   'CMPTR  conv_1 cm_rc'
   x = '   Prepare_To_Receive (CMPTR) ........'
   call Display x 'return_code =' cm_return_code.cm_rc
   if (cm_rc ¬= cm_ok) then call Error 'Unexpected return_code on CMPTR'

   'CMCFM  conv_1 rts_received cm_rc'
   x = '   Confirm (CMCFM) ...................'
   call Display x 'return_code =' cm_return_code.cm_rc
   if (cm_rc ¬= cm_ok) then call Error 'Unexpected return_code on CMCFM'

   call Display ' '
   x = '   Waiting for CRREXMP2 to decide COMMIT or BACKOUT.'
   call Display x
```

```
   call Display ' '
   'CMRCV  conv_1 buf 10 data_rec rec_len stat_rec rts_rec cm_rc'
   x = '   Receive (CMRCV) ..................'
   call Display x 'return_code =' cm_return_code.cm_rc
   x = '                                    '
   call Display x 'status_rec  =' cm_status_received.stat_rec
   if (cm_rc ¬= cm_ok) & (cm_rc ¬= cm_take_backout) then,
      call Error 'Unexpected return_code on CMRCV'
   if (cm_rc = cm_ok) & (stat_rec ¬= cm_take_commit_deallocate) then,
      call Error 'Unexpected status_received on CMRCV'

   if (cm_rc = cm_take_backout) then do
      call Issue_Backout
      call Receive_Deallocation
   end
      else call Choose_Syncpoint

   return
/*--------------------------------------------------------------------*/
/*  Following are the sub-routines called throughout this EXEC.       */
/*--------------------------------------------------------------------*/

/*--------------------------------------------------------------------*/
/*  CHOOSE_SYNCPOINT:                                                 */
/*                                                                    */
/*  For demonstration purposes, the user is allowed to choose         */
/*  whether to commit or backout the changes once the user has        */
/*  received the notification that a commit has been requested.       */
/*--------------------------------------------------------------------*/

Choose_Syncpoint:

   call Display ' '
   call Display ' '
   x = 'CRREXMP1:  Choosing between COMMIT or BACKOUT:'
   call Display x
   call Display ' '
   resp = ''
   x = '   Enter COMMIT to commit, anything else to backout.'
   call Display x
   pull choice
   if (choice = 'COMMIT') then do
      call Display ' '
      x = '   COMMIT chosen by user.'
      call Display x
      call Display ' '
      call Issue_Commit
   end
      else do
         call Display ' '
         x = '   BACKOUT chosen by user.'
         call Display x
         call Display ' '
         call Issue_Backout
         call Receive_Deallocation
      end

   return

/*--------------------------------------------------------------------*/
/*  ISSUE_COMMIT:                                                     */
/*                                                                    */
/*  This sub-routine will process commit by issuing SRRCMIT.          */
/*  If the SRRCMIT is not successful, error processing is             */
/*  performed.                                                        */
/*--------------------------------------------------------------------*/

Issue_Commit:

   address cpirr
   'SRRCMIT rr_rc'
   x = '   Commit (SRRCMIT) ..................'
   call Display x 'return_code =' rr_return_code.rr_rc

   if (rr_rc ¬= rr_ok) then do
      if (rr_rc >= 300) then commit_turned_backout = 1
      call Error 'Unexpected return_code on SRRCMIT'
   end

   call Display ' '
   x = '   <<<  UPDATES IN PLACE, COMMIT PERFORMED  >>>'
   call Display x
```

```
   return

/*-------------------------------------------------------------------*/
/*  ISSUE_BACKOUT:                                                   */
/*                                                                   */
/*  This sub-routine will handle backout processing by issuing       */
/*  SRRBACK.  If a previous commit attempt resulted in a backout     */
/*  return code, then the SRRBACK is not performed because the       */
/*  changes have already been backed out.                            */
/*-------------------------------------------------------------------*/

Issue_Backout:

   if ¬commit_turned_backout then do
      address cpirr
      'SRRBACK rr_rc'
      x = '   Backout (SRRBACK) ................'
      call Display x 'return_code =' rr_return_code.rr_rc
      if (rr_rc ¬= rr_ok) then                              ,
         call Error 'Unexpected return_code on SRRBACK'
   end

   if conversation_active then do
      call Display ' '
      x = '   <<<  UPDATES NOT MADE, BACKOUT PERFORMED  >>>'
      call Display x
      call Display ' '
   end

   return


/*-------------------------------------------------------------------*/
/*  RECEIVE_DEALLOCATION:                                            */
/*                                                                   */
/*  This sub-routine will issue CMRCV to receive the deallocation    */
/*  issued by the partner.  If the return_code ends with '_BO',      */
/*  then we need to issue a backout.                                 */
/*-------------------------------------------------------------------*/

Receive_Deallocation:

   address cpicomm
   'CMRCV  conv_1 buf 10 data_rec rec_len stat_rec rts_rec cm_rc'
   x = '   Receive (CMRCV) ..................'
   call Display x 'return_code =' cm_return_code.cm_rc
   conversation_active = 0

   if (right(cm_return_code.cm_rc, 3) = '_BO') then call Issue_Backout

   return


/*-------------------------------------------------------------------*/
/*  ISSUE_DEALLOCATE_ABEND:                                          */
/*                                                                   */
/*  This sub-routine will issue CMSDT to set the cm_deallocate_type  */
/*  to cm_deallocate_abend, then issue CMDEAL to deallocate the      */
/*  conversation.                                                    */
/*-------------------------------------------------------------------*/

Issue_Deallocate_Abend:

   address cpicomm
   'CMSDT  conv_1 cm_deallocate_abend cm_rc'
   x = '   Set_Deallocate_Type (CMSDT) .......'
   call Display x 'return_code =' cm_return_code.cm_rc

   'CMDEAL conv_1 cm_rc'
   x = '   Deallocate (CMDEAL) ..............'
   call Display x 'return_code =' cm_return_code.cm_rc
   conversation_active = 0

   return


/*-------------------------------------------------------------------*/
/*  ERROR:                                                           */
/*                                                                   */
/*  This sub-routine will display an error message and exit.  A      */
/*  backout is performed to roll back any outstanding work that has  */
```

```
/*  not been committed.  If a CPI Communications conversation is    */
/*  active, a deallocate is done on that conversation with          */
/*  cm_deallocate_type of cm_deallocate_abend.                      */
/*------------------------------------------------------------------*/

Error:

   parse arg err_msg

   errors_found = errors_found + 1
   call Display ''
   err_msg = 'CRREXMP1 Error: ' err_msg
   call Display ''
   call Display err_msg
   call Display ''

   if (errors_found = 1) then do
      call Issue_Backout
      if conversation_active then call Issue_Deallocate_Abend
      signal CRREXMP1_Exit
   end

   return


/*------------------------------------------------------------------*/
/*  DISPLAY:                                                        */
/*                                                                  */
/*  This routine will display an input message to the virtual       */
/*  machine console, and use EXECIO to write the message to a file. */
/*------------------------------------------------------------------*/
Display:

   parse arg disp_msg
   say disp_msg
   address cms 'execio 1 diskw CRREXMP1 CONSOLE A (STRING' disp_msg

   return
```

Target Application, CRREXMP2 EXEC

The following is the target REXX application, CRREXMP2 EXEC:

```
/*------------------------------------------------------------------*/
/*  CRREXMP2 EXEC -- Target application                             */
/*------------------------------------------------------------------*/

/*------------------------------------------------------------------*/
/*  Main Logic of CRREXMP2:                                         */
/*                                                                  */
/*     Initialize program constants                                 */
/*     Verify the required SFS directory exists                     */
/*     Receive a request to update a file                           */
/*     Set synchronization point options and transaction tag        */
/*     Update the file                                              */
/*     Perform synchronization point processing                     */
/*------------------------------------------------------------------*/


call Initialize
call Verify_SFS
call Get_Request
call Setup_Syncpoint
call Update_File file3, dir3

call Choose_Syncpoint

CRREXMP2_Exit:

   call Display ' '
   call Display 'CRREXMP2 Complete.'
   call Display ' '

   exit
/*------------------------------------------------------------------*/
/*  INITIALIZE:                                                     */
/*                                                                  */
/*  This sub-routine will set up various constants and variables    */
/*  that will be used throughout this EXEC.                         */
```

```
/*                                                                   */
/*  NOTES:                                                           */
/*          -  Variable dir3 must be set to an SFS directory that    */
/*             the userid running this EXEC owns or has write        */
/*             access to.  It cannot be in the same filepool as      */
/*             the directories the CRREXMP1 EXEC uses.               */
/*                                                                   */
/*          -  Files SRRREXX COPY and CMREXX COPY provide constants  */
/*             and literals for SAA resource recovery and CPI        */
/*             Communications.  They must exist on an accessed       */
/*             minidisk or SFS directory.                            */
/*-------------------------------------------------------------------*/

Initialize:

   'vmfclear'

   dash               = '*' || copies('-', 68) || '*'
   blanks25           = copies(' ', 25)
   conversation_active = 0
   error_found        = 0

   call Display dash
   x = 'CRREXMP2:  Sample Coordinated Resource Recovery Application'
   call Display x
   call Display dash
   call Display ' '

   file3 = 'SANTAS SACK'
   dir3  = 'SERVER5:HEALDJS.CRRDIR3.'

   copy_file.1 = 'CMREXX COPY *'
   copy_file.2 = 'SRRREXX COPY *'
   do i = 1 to 2
      address command 'ESTATE' copy_file.i
      if (rc ¬=0) then call Error copy_file.i 'file not found'
         else do
            'execio * diskr' copy_file.i '(finis stem PSEUDONYM.'
            do j = 1 to pseudonym.0
               interpret pseudonym.j
            end
         end
   end

   return

/*-------------------------------------------------------------------*/
/*  VERIFY_SFS                                                       */
/*                                                                   */
/*  This sub-routine will verify that required SFS directories       */
/*  are available for the EXEC to run.                               */
/*-------------------------------------------------------------------*/

Verify_SFS:

   x = 'CRREXMP2:  Verifying required directories are available.'
   call Display x
   call Display ' '

   len1 = length(dir3)
   call csl('DMSEXIDI exidi_rc exidi_rs dir3 len1 NOCOMMIT 8')
   erc = overlay(exidi_rc, '          ')
   x = '           DMSEXIDI   Retcode =' erc  'Reascode =' exidi_rs
   call Display x
   if (exidi_rc ¬= 0) then call Error 'Unexpected result on DMSEXIDI'
   call Display ' '

   return


/*-------------------------------------------------------------------*/
/*  GET_REQUEST:                                                     */
/*                                                                   */
/*  This sub-routine uses CPI Communications routines to             */
/*  communicate with CRREXMP1 and receive the transaction data       */
/*  to be added to the file managed by CRREXMP2.  The following      */
/*  routines are used:                                               */
/*                                                                   */
/*  CMACCP ... accept conversation allocated by CRREXMP1             */
/*  CMRCV  ... receive confirmation request                          */
/*  CMCFMD ... respond to the confirmation request                   */
/*  CMRCV  ... receive transaction data and confirmation request     */
```

```
/*             to change to send state                              */
/*  CMCFMD ... confirm change to send state so we can initiate      */
/*             syncpoint processing                                 */
/*-----------------------------------------------------------------*/

Get_Request:

   call Display ' '
   x = 'CRREXMP2:  Getting transaction request on protected conversation'
   call Display x
   call Display ' '

   address cpicomm
   'CMACCP conv_1 cm_rc'
   x = '   Accept_Conversation (CMACCP) ......'
   call Display x 'return_code =' cm_return_code.cm_rc
   if (cm_rc ¬= cm_ok) then call Error 'Unexpected return_code on CMACCP'

   conversation_active = 1

   'CMRCV  conv_1 buf 10 data_rec rec_len stat_rec rts_rec cm_rc'
   x = '   Receive (CMRCV) ..................'
   call Display x 'return_code =' cm_return_code.cm_rc
   if (cm_rc ¬= cm_ok) then call Error 'Unexpected return_code on CMRCV'
   x = '                                    '
   call Display x 'status_rec  =' cm_status_received.stat_rec
   if (stat_rec ¬= cm_confirm_received) then                        ,
      call Error 'Unexpected status_received on CMRCV'

   'CMCFMD conv_1 cm_rc'
   x = '   Confirmed (CMCFMD) ................'
   call Display x 'return_code =' cm_return_code.cm_rc
   if (cm_rc ¬= cm_ok) then call Error 'Unexpected return_code on CMCFMD'

   'CMRCV  conv_1 buf 90 data_rec rec_len stat_rec rts_rec cm_rc'
   x = '   Receive (CMRCV) ..................'
   call Display x 'return_code =' cm_return_code.cm_rc
   if (cm_rc ¬= cm_ok) then call Error 'Unexpected return_code on CMRCV'
   x = '                                    '
   call Display x 'data_rec    =' cm_data_received.data_rec
   if (data_rec ¬= cm_complete_data_received) then                   ,
      call Error 'Unexpected data_received on CMRCV'
   tran = strip(buf)
   tran_length = length(tran)

   if (stat_rec = cm_no_status_received) then do
      'CMRCV  conv_1 buf 10 data_rec rec_len stat_rec rts_rec cm_rc'
      x = '   Receive (CMRCV) ..................'
      call Display x 'return_code =' cm_return_code.cm_rc
      if (cm_rc ¬= cm_ok) then                                     ,
         call Error 'Unexpected return_code on CMRCV'
   end

   call Display x 'status_rec  =' cm_status_received.stat_rec
   if (stat_rec ¬= cm_confirm_send_received) then             ,
      call Error 'Unexpected status_received on CMRCV'
   'CMCFMD conv_1 cm_rc'
   x = '   Confirmed (CMCFMD) ................'
   call Display x 'return_code =' cm_return_code.cm_rc
   if (cm_rc ¬= cm_ok) then
      call Error 'Unexpected return_code on CMCFMD'

   return


/*-----------------------------------------------------------------*/
/*  SETUP_SYNCPOINT:                                               */
/*                                                                 */
/*  This sub-routine sets some options for synchronization points:  */
/*  First, DMSSSPTO is called to indicate that if resynchronization */
/*  is required, control will be returned to the application if it  */
/*  is not possible to complete the resynchronization immediately.  */
/*                                                                 */
/*  Next, DMSSETAG is called to set the transaction tag for this    */
/*  transaction.  CMS stores this tag on the log to assist an       */
/*  operator in the event that they need to intervene during        */
/*  resynchronization.  In this case, the actual data record that   */
/*  we are attempting to add to the SFS files is used as the tag.   */
/*                                                                 */
/*  An error from either of these routines is displayed, but        */
/*  processing will continue.                                      */
/*-----------------------------------------------------------------*/
```

```
Setup_Syncpoint:

   call Display ' '
   call Display ' '
   x = 'CRREXMP2:  Setting Synchronization Point Options'
   call Display x
   call Display ' '
   call csl('DMSSSPTO sspto_rc sspto_rs NOWAIT 6')
   src = overlay(sspto_rc, '          ')
   x = '           DMSSSPTO   Retcode =' src  'Reascode =' sspto_rs
   call Display x

   call Display ' '
   call Display ' '
   x = 'CRREXMP2:  Setting Transaction Tag'
   call Display x
   call Display ' '
   call csl('DMSSETAG setag_rc setag_rs tran tran_length')
   src = overlay(setag_rc, '          ')
   x = '           DMSSETAG   Retcode =' src  'Reascode =' setag_rs
   call Display x
   x = '           Tag =' tran
   call Display x

   return


/*--------------------------------------------------------------------*/
/*  UPDATE_FILE:                                                      */
/*                                                                    */
/*  This sub_routine updates a file in an SFS directory by adding     */
/*  one record.  The file and SFS directory are input to this         */
/*  sub-routine.  If the file is successfully opened and updated,     */
/*  it is then closed with the NOCOMMIT option.  CSL routines         */
/*  issued include:                                                   */
/*                                                                    */
/*  DMSOPEN  ... open the input file in the input directory           */
/*  DMSWRITE ... write the input data record to the open file         */
/*  DMSCLOSE ... close the input file in the input directory          */
/*--------------------------------------------------------------------*/

Update_File:

   arg file, dir

   call Display ' '
   call Display ' '
   call Display 'CRREXMP2:  UPDATING <'file'> in <'dir'>'
   call Display ' '

   file_spec = file dir
   len1 = length(file_spec)

   call csl('DMSOPEN open_rc open_rs file_spec len1 WRITE 5 token')
   orc = overlay(open_rc, '          ')
   x = '           DMSOPEN    Retcode =' orc  'Reascode =' open_rs
   call Display x

   if (open_rc = 0) | ((open_rc = 4) & (open_rs = 44030)) then do
      call csl('DMSWRITE write_rc write_rs token 1 tran_length tran 80')
      wrc = overlay(write_rc, '          ')
      x = '           DMSWRITE   Retcode =' wrc 'Reascode =' write_rs
      call Display x
      if (write_rc = 0) then do
         call csl('DMSCLOSE close_rc close_rs token NOCOMMIT 8')
         crc = overlay(close_rc, '          ')
         x = '           DMSCLOSE   Retcode =' crc  'Reascode =' close_rs
         call Display x
         if (close_rc ¬= 0) then                                    ,
            call Error 'Unexpected result on DMSCLOSE'
      end

         else call Error 'Unexpected result on DMSWRITE'
   end

      else call Error 'Unexpected result on DMSOPEN'


   return


/*--------------------------------------------------------------------*/
```

```
/*  CHOOSE_SYNCPOINT:                                                 */
/*                                                                    */
/*  This sub_routine asks the user to choose whether to commit or     */
/*  backout all changes for the current workunit.  Our current        */
/*  workunitid is associated with CRREXMP1's current workunitid       */
/*  because they have the same logical unit of work identifier        */
/*  (LUWID).  The work involved in this syncpoint then includes:      */
/*                                                                    */
/*     -  CRREXMP1 updates to file CHILDS   LIST   in .CRRDIR1        */
/*     -  CRREXMP1 updates to file TOYSTORE ORDERS in .CRRDIR2        */
/*     -  CRREXMP2 updates to file SANTAS   SACT   in .CRRDIR3        */
/*                                                                    */
/*  The protected conversation established between CRREXMP1 and       */
/*  CRREXMP2 will permit syncpoint requests for the common LUWID      */
/*  to be processed.                                                  */
/*                                                                    */
/*  By performing a commit, a request will be sent to CRREXMP1 on     */
/*  the protected conversation to commit the updates to CHILDS LIST   */
/*  and TOYSTORE ORDERS.   If CRREXMP1 responds by issuing commit,    */
/*  updates to its files as well as those made to SANTAS SACK by      */
/*  CRREXMP will be committed.  If CRREXMP1 responds by issuing       */
/*  backout, then updates to all the files are rolled back.           */
/*                                                                    */
/*  By performing a backout, a request will be sent to CRREXMP1 on    */
/*  the protected conversation to roll back the updates to CHILDS     */
/*  LIST and TOYSTORE ORDERS.  CRREXMP1 will issue SRRBACK to do      */
/*  this backout, and the updates to SANTAS SACK made by CRREXMP2     */
/*  will then be rolled back as well.                                 */
/*--------------------------------------------------------------------*/
Choose_Syncpoint:

   call Display ' '
   call Display ' '
   x = 'CRREXMP2:  Choosing between COMMIT or BACKOUT:'
   call Display x
   call Display ' '

   x = '   Enter COMMIT to commit, anything else to backout.'
   call Display x
   pull choice
   if (choice = 'COMMIT') then do
      call Display ' '
      x = '   COMMIT chosen by user.'
      call Display x
      call Display ' '
      call Issue_Commit
   end

      else do
         call Display ' '
         x = '   BACKOUT chosen by user.'
         call Display x
         call Display ' '
         call Issue_Backout
         call Issue_Deallocate_Abend
      end

   return
/*--------------------------------------------------------------------*/
/*  Following are the sub-routines called throughout this EXEC.       */
/*--------------------------------------------------------------------*/

/*--------------------------------------------------------------------*/
/*  ISSUE_COMMIT:                                                     */
/*                                                                    */
/*  This sub-routine will process commit by issuing SRRCMIT.  If      */
/*  the commit fails, then backout processing will be called.         */
/*--------------------------------------------------------------------*/

Issue_Commit:

   call Deallocate_Sync_Level
   address cpirr
   'SRRCMIT rr_rc'
   x = '   Commit (SRRCMIT) ..................'
   call Display x 'return_code =' rr_return_code.rr_rc

   if (rr_rc >= 300) then do
      x = '   <<<  UPDATES NOT MADE, COMMIT TURNED TO BACKOUT >>>'
      call Display ' '
      call Display x
      call Display ' '
```

```
      call Issue_Deallocate_Abend
   end
      else do
         x = '   <<<  UPDATES IN PLACE, COMMIT PERFORMED  >>>'
         call Display ' '
         call Display x
         call Display ' '
      end

   return


/*--------------------------------------------------------------------*/
/*  ISSUE_BACKOUT:                                                    */
/*                                                                    */
/*  This sub-routine will initiate backout processing by issuing    */
/*  SRRBACK.                                                          */
/*--------------------------------------------------------------------*/

Issue_Backout:

   address cpirr
   'SRRBACK rr_rc'
   x = '   Backout (SRRBACK) ................'
   call Display x 'return_code =' rr_return_code.rr_rc
   if (rr_rc ¬= rr_ok) then                                          ,
      call Error 'Unexpected return code on SRRBACK'
   if conversation_active then do
      call Display ' '
      x = '   <<<  UPDATE NOT MADE, BACKOUT PERFORMED  >>>'
      call Display x
      call Display ' '
   end

   return


/*--------------------------------------------------------------------*/
/*  DEALLOCATE_SYNC_LEVEL:                                            */
/*                                                                    */
/*  This sub_routine issues CMDEAL to deallocate the protected      */
/*  conversation.  The default cm_deallocate_type of                  */
/*  cm_deallocate_sync_level is used, which means the conversation  */
/*  will not be deallocated until after the next successful         */
/*  synchronization point.                                            */
/*--------------------------------------------------------------------*/

Deallocate_Sync_Level:

   address cpicomm
   'CMDEAL conv_1 cm_rc'
   x = '   Deallocate (CMDEAL) ..............'
   call Display x 'return_code =' cm_return_code.cm_rc
   if (cm_rc ¬= cm_ok) then
       call Error 'Unexpected return_code on CMDEAL'

   return


/*--------------------------------------------------------------------*/
/*  ISSUE_DEALLOCATE_ABEND:                                           */
/*                                                                    */
/*  This sub_routine issues CMSDT to set the cm_deallocate_type to  */
/*  cm_deallocate_abend, then issues CMDEAL to deallocate the       */
/*  protected conversation.                                           */
/*--------------------------------------------------------------------*/

Issue_Deallocate_Abend:

   address cpicomm
   'CMSDT  conv_1 cm_deallocate_abend cm_rc'
   x = '   Set_Deallocate_Type (CMSDT) .......'
   call Display x 'return_code =' cm_return_code.cm_rc
   if (cm_rc ¬= cm_ok) then
       call Error 'Unexpected return_code on CMSDT'
   'CMDEAL conv_1 cm_rc'
   x = '   Deallocate (CMDEAL) ..............'
   call Display x 'return_code =' cm_return_code.cm_rc
   if (cm_rc ¬= cm_ok) then
       call Error 'Unexpected return_code on CMDEAL'

   return
```

```
/*--------------------------------------------------------------------*/
/*  DISPLAY:                                                          */
/*                                                                    */
/*  This routine will display an input message to the virtual         */
/*  machine console, and use EXECIO to write the message to a file.   */
/*--------------------------------------------------------------------*/

Display:

   parse arg disp_msg
   say disp_msg
   address cms 'execio 1 diskw CRREXMP2 CONSOLE A (STRING' disp_msg

   return


/*--------------------------------------------------------------------*/
/*  ERROR:                                                            */
/*                                                                    */
/*  This sub_routine will display an error message and exit.  A       */
/*  backout is performed to roll back any outstanding work that has   */
/*  not been committed.  If a CPI Communications conversation is      */
/*  active, a deallocate is done on that conversation with            */
/*  cm_deallocate_type of cm_deallocate_abend.                        */
/*--------------------------------------------------------------------*/
Error:

   parse arg err_msg

   errors_found = 1

   err_msg = 'CRREXMP2 Error: ' err_msg
   call Display ''
   call Display err_msg
   call Display ''

   if (errors_found = 1) then do
      call Issue_Backout
      if conversation_active then call Issue_Deallocate_Abend
      signal CRREXMP2_Exit
   end

   return
```

Appendix I. CRR Communications Examples

These CRR Communications examples follow:

These are examples of CRR processing for resource managers participating in CRR. This information is provided for programmers with product development responsibilities.

Note: The scenarios describe the communication between the resource adapter and the resource manager and the communication between the resource manager and the CRR recovery server. The examples in the scenarios use APPC/VM assembler macros. The scenarios do not provide examples of protected conversations.

Single Processor Case

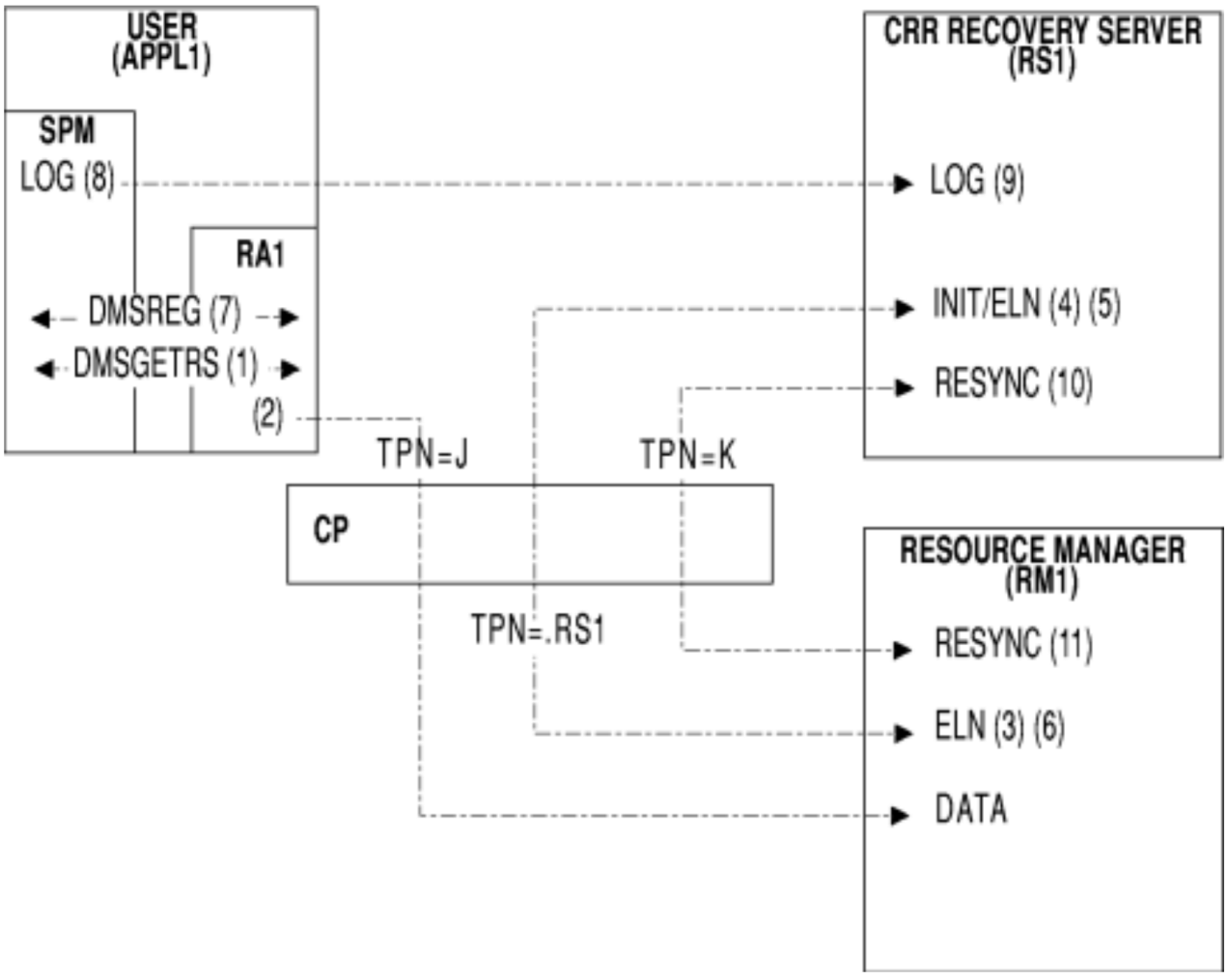


Figure 111. CRR Communications on a Single Processor

1. User virtual machine APPL1 is running a CMS application that uses the resource managed by resource manager RM1.

Resource adapter RA1, RM1's adapter in the APPL1 virtual machine, calls the CSL routine DMSGETRS to get the CRR recovery server's current log name and TPN from the CRR sync point manager (SPM). In this example the values are CRR25 and .RS1, respectively.

2. Because CMS communications directory (COMDIR) resolved information is needed to register the resource for CRR, the resource adapter should issue an explicit resolve for the resource manager. In this example, assume that the resource manager's COMDIR name resolves to locally known LU name *IDENT 0, TPN J, and mode name 0.

   The resource adapter then establishes a conversation with (connects to) the resource manager at locally known LU name *IDENT 0, TPN J, mode name 0 and sends the CRR recovery server's log name and TPN in a data buffer. (The resource adapter sends this information in a data buffer each time it connects to the resource manager.)

   When the resource adapter initiates the connection, the resource manager receives connection pending extended data (allocate data). Before the resource manager accepts the connection, it must get certain information from this data. (The connection pending extended data is not saved after the resource manager accepts the connection.)

   Among the parameters in the connection pending extended data are the sender's (resource adapter's) locally known LU name (*USERID APPL1 in this example), the mode name (0), and the local (resource manager's) fully qualified LU name (0). The locally known LU name in the connection pending extended data is sometimes called the "connect back" locally known LU name because it is used for connecting back to that LU. The mode name and local fully qualified LU name are both 0 because these parameters are not used when the partners are on the same processor.

   To connect to the CRR recovery server, the resource manager needs to determine the CRR recovery server's locally known LU name. Because the resource adapter and the CRR recovery server are always on the same processor, the resource manager can use the "connect back" locally known LU name from the connection pending extended data. The resource manager interprets the LU name qualifier *USERID to mean that the resource manager must use *IDENT 0 as the locally known LU name for connecting to that LU.

   The resource manager stores the locally known LU name, mode name, and local fully qualified LU name to use later. However, at this point the resource manager does *not* record any data in its log name log.

   After the resource manager accepts the connection, the resource adapter receives connection complete extended data. From this data, the resource adapter must save the local (resource adapter's) fully qualified LU name (0) and the remote (resource manager's) fully qualified name (0) to use in the CRR registration.

3. The resource manager receives the CRR recovery server's log name (CRR25) and TPN (.RS1) in the data buffer from the resource adapter and determines if an exchange of log names with the CRR recovery server is required. The resource manager looks for:

   a. A record in its log name log.

      The resource manager uses locally known LU name *IDENT 0, TPN .RS1, and log name CRR25 as search arguments. In this example, we are assuming that there is no matching record.

      The resource manager also uses the CRR recovery server's locally known LU name and TPN to determine the status of the resource manager's log if an initial Exchange Log Names message is sent. If both names match the log, the log is warm. If either name does not match the log or is missing, the log is cold.

   b. A local indication that log names were exchanged.

      This indication can be a local flag associated with each log name record, a local caching of the log name record, or some other method. Whatever technique is chosen, its purpose is to determine if the resource manager has exchanged log names with the CRR recovery server during the resource manager's current activation.

      In cases where a resource manager could accidentally use the wrong log (as opposed to intentionally cold-logging), such as when logs are mounted or archived, it is important to force

an exchange of log names with the CRR recovery server at least once for each activation of the resource manager to:

- Catch a warm/warm log mismatch, where it is possible that either the CRR recovery server's log or the resource manager's log is the wrong one.
- Avoid having to check for this warm/warm mismatch on every connect—just once each time the resource manager is activated.
- Accomplish an exchange of log names in case the CRR recovery server has erased log entries for the resource without cold-logging.

In our example, because there is no matching record in the resource manager's log and no local indication of a previous exchange, an exchange is required. The resource manager formulates an Exchange Log Names request that includes the following data:

Parameter	Value	Meaning
log status flag	X'00'	COLD—the CRR recovery server's locally known LU name and TPN were not known to the resource manager (not found in its log).
local fully qualified LU name	0	The resource manager has no local fully qualified LU name.
TPN	J	This is the resource manager's TPN.
log name 1	RESLOG5	This is the resource manager's log name.

The resource manager then connects to the CRR recovery server at locally known LU name *IDENT 0, TPN .RS1, mode name 0 and sends the Exchange Log Names message.

4. The CRR recovery server recognizes TPN .RS1 on the incoming connection as the special case of an Exchange Log Names request from a participating resource manager.

   In the connection from the resource manager, the CRR recovery server receives connection pending extended data (allocate data). This data includes the sender's (resource manager's) locally known LU name (*USERID RM1), the local (CRR recovery server's) fully qualified LU name (0), and the remote (resource manager's) fully qualified LU name (0). The CRR recovery server interprets the qualifier *USERID in the locally known LU name to mean that the CRR recovery server must use locally known LU name *IDENT 0 for connecting back to the resource manager. The fully qualified LU names are both 0 because these parameters are not used when the partners are on a single processor.

   The CRR recovery server receives the Exchange Log Names message, which contains the resource manager's TPN (J) and log name (RESLOG5). The CRR recovery server then looks for locally known LU name *IDENT 0, remote fully qualified LU name 0, TPN J, and log name (RESLOG5) in its log name table. In our example, the CRR recovery server finds no match for this combination, so it adds a record, as follows:

Table 72. Example of a Log Name Table Record in the CRR Recovery Server's Log, Single Processor Case

LNLU	TPN	LOC FQLU	REM FQLU	LOGNAME
*IDENT 0	J	0	0	RESLOG5

5. The CRR recovery server formulates an Exchange Log Names reply that includes the following data and sends it to the resource manager:

Parameter	Value	Meaning
log status flag	X'00'	COLD—the resource manager's locally known LU name and TPN were not known to the CRR recovery server (not found in the log name table).

Parameter	Value	Meaning
local fully qualified LU name	0	The CRR recovery server has no local fully qualified LU name.
TPN	.RS1	This is the CRR recovery server's TPN.
log name 1	CRR25	This is the CRR recovery server's log name.

6. The resource manager receives the Exchange Log Names reply. Taking the CRR recovery server's log name and TPN from the message, together with the CRR recovery server's locally known LU name and the local (resource manager's) fully qualified LU name captured previously (Step 2), the resource manager adds a record to its log name log, as follows:

Table 73. Example of a Log Name Record in the Resource Manager's Log, Single Processor Case

LNLU	TPN	LOC FQLU	LOGNAME
*IDENT 0	.RS1	0	CRR25

Note: Both LU names are important for a possible "shoulder tap", if the resource manager needs to notify the CRR recovery server of the resource manager's presence and readiness to accept resynchronization communications. The resource manager uses the locally known LU name and the TPN to connect to the CRR recovery server. The resource manager then sends its own fully qualified LU name and log name in an Exchange Log Names request.

7. The resource adapter calls the CSL routine DMSREG to register with the SPM. Using this routine, the resource adapter provides the resource TPN (J), the resource recovery TPN (K), a recovery token, the mode name (0), the local fully qualified LU name (0), and the remote fully qualified LU name (0), along with other parameters.

   Note: The resource recovery TPN (K) is optional, and is assigned by the resource manager. In case of a failure during a sync point, the CRR recovery server uses the resource recovery TPN, if defined, to reach the resource manager. This lets the resource manager know that the connection is for resynchronization and not a normal data request. If a resource recovery TPN is not provided, the resource adapter must specify PIP data (also defined by the resource manager) in the registration. In that case, the CRR recovery server uses the resource TPN (J in this example) to reach the resource manager, but includes the PIP data on the connect to indicate that the connection is for resynchronization.

8. The SPM stores (for subsequent logging) the following information to represent the resource:
   - Resource TPN (J)
   - Local fully qualified LU name (0)
   - Remote fully qualified LU name (0)
   - Mode name (0)
   - Resource recovery TPN (K)
   - Recovery token
   - Other parameters not covered in this example.
9. At the start of a commit, the CRR recovery server receives a write log request from the SPM and writes an SPM Pending log record in the CRR recovery server's log. This log record contains the resource information listed in the previous step.
10. Assume that a failure occurs during a sync point for this application and that resynchronization recovery is started to recover this resource. The SPM Pending record for this resource in the CRR recovery server's log contains the following information:

Table 74. Example of an SPM Pending Record in the CRR Recovery Server's Log, Single Processor Case

LOC FQLU	REM FQLU	LUWID	TPN	REC TPN	REC TOKEN	MODENAME
0	0		J	K		0

Note: The LUWID and recovery token values are not shown only because they are not critical in this example, which is concentrating on LUs, TPNs, log names, and so on.

The CRR recovery server uses local fully qualified LU name 0, remote fully qualified LU name 0, and TPN J to search the log name table in its log for a matching record (see Table 72 on page 585). From that entry, the CRR recovery server gets the resource manager's locally known LU name, *IDENT 0.

Using information from the SPM Pending record and the log name table, the CRR recovery server connects to the resource manager at locally known LU name *IDENT 0, TPN K, mode name 0.

The CRR recovery server formulates an Exchange Log Names request message that includes the following data and sends it to the resource manager:

Parameter	Value	Meaning
log status flag	X'01'	WARM—the CRR recovery server has a record in its log name table that contains the combination of local fully qualified LU name 0, remote fully qualified LU name 0, and TPN J. Note that the CRR recovery server's log is always warm in a resynchronization recovery Exchange Log Names request.
local fully qualified LU name	0	The CRR recovery server has no local fully qualified LU name.
TPN	.RS1	This is the CRR recovery server's TPN.
log name 1	CRR25	This is the CRR recovery server's log name.
log name 2	RESLOG5	This is the resource manager's log name that the CRR recovery server has saved in its log.

In the same buffer as the Exchange Log Names request, the CRR recovery server sends a Compare States request that identifies the LUWID and the current state of the logical unit of work.

11. The resource manager recognizes TPN K on the incoming connection as a special case, a resynchronization request from the CRR recovery server.

    The connection pending extended data (allocate data) that the resource manager receives contains the CRR recovery server's locally known LU name, *USERID RS1. The resource manager interprets this to mean that the resource manager must use locally known LU name *IDENT 0 when searching its log for a record for the CRR recovery server.

    After accepting the connection, the resource manager receives the Exchange Log Names message, which contains the CRR recovery server's TPN (.RS1) and log name (CRR25). The resource manager then uses locally known LU name *IDENT 0 and TPN .RS1 to search for a matching log name record in its log. In our example, the resource manager finds such a record (see Table 73 on page 586) and verifies that the associated log name saved in the log (CRR25) matches the log name sent in the message. The resource manager can also verify that its own log name sent in the message (RESLOG5) is correct.

    The resource manager formulates an Exchange Log Names reply to confirm that everything matches. The reply includes the following data:

Parameter	Value	Meaning
log status flag	X'01'	WARM—the resource manager has previously communicated with the CRR recovery server.
local fully qualified LU name	0	The resource manager has no local fully qualified LU name.
TPN	K	This is the resource manager's recovery TPN.
log name 1	RESLOG5	This is the resource manager's log name.

The resource manager holds the Exchange Log Names reply and sends it to the CRR recovery server in the same buffer as the Compare States reply.

Note: The resolution of the Compare States request is not covered in this example.

12. The CRR recovery server processes the Exchange Log Names and Compare States replies and severs the connection to (deallocates the conversation with) the resource manager.

TSAF Collection Case

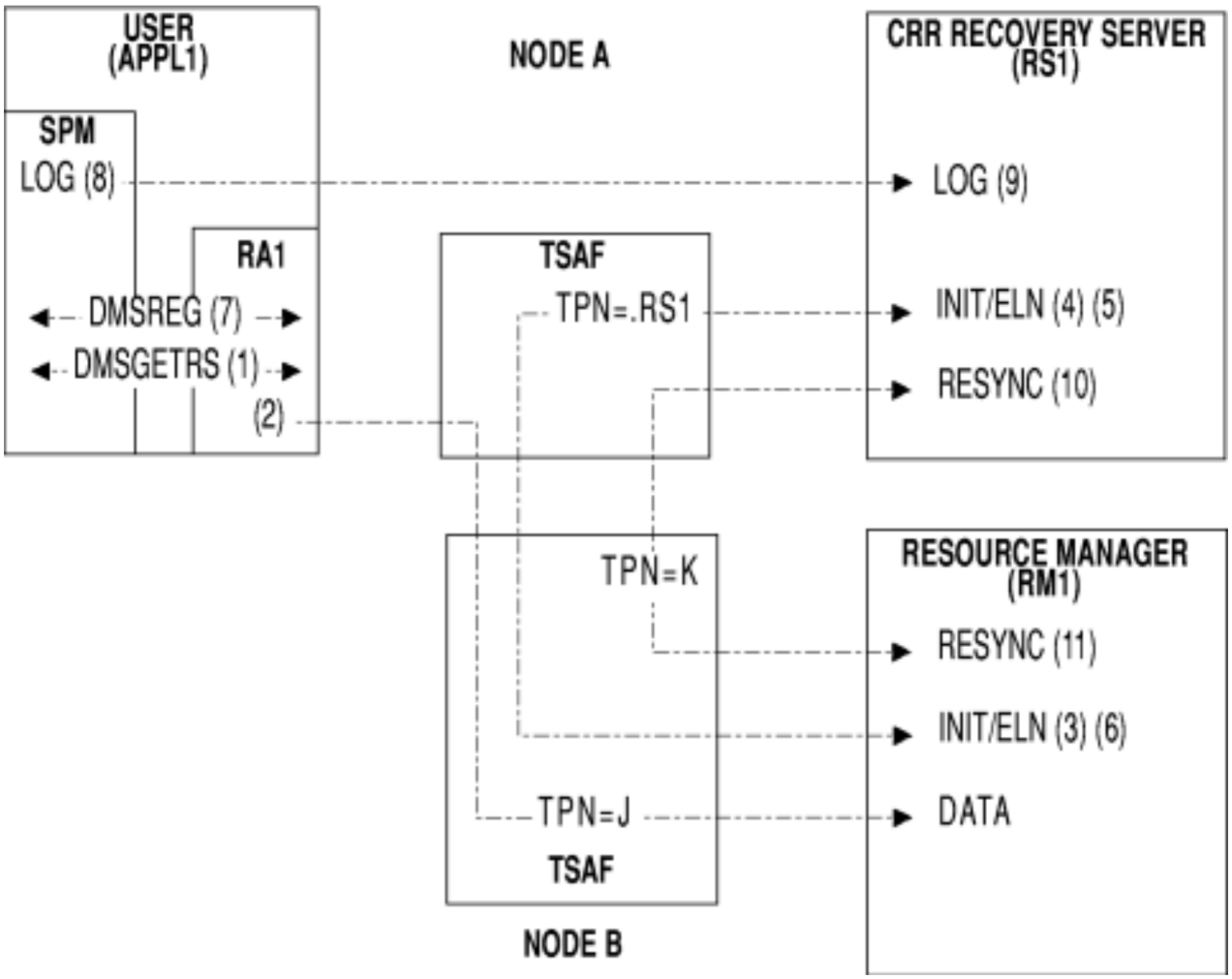


Figure 112. CRR Communications in a TSAF Collection

1. User virtual machine APPL1 is running a CMS application that uses the resource managed by resource manager RM1.

Resource adapter RA1, RM1's adapter in the APPL1 virtual machine, calls the CSL routine DMSGETRS to get the CRR recovery server's current log name and TPN from the CRR sync point manager (SPM). In this example the values are CRR25 and .RS1, respectively.

2. Because CMS communications directory (COMDIR) resolved information is needed to register the resource for CRR, the resource adapter should issue an explicit resolve for the resource manager. In this example, assume that the resource manager's COMDIR name resolves to locally known LU name *IDENT 0, TPN J, and mode name 0.

   The resource adapter then establishes a conversation with (connects to) the resource manager at locally known LU name *IDENT 0, TPN J, mode name 0 and sends the CRR recovery server's log name and TPN in a data buffer. (The resource adapter sends this information in a data buffer each time it connects to the resource manager.)

   When the resource adapter initiates the connection, the resource manager receives connection pending extended data (allocate data). Before the resource manager accepts the connection, it must get certain information from this data. (The connection pending extended data is not saved after the resource manager accepts the connection.)

   Among the parameters in the connection pending extended data are the sender's (resource adapter's) locally known LU name (*USERID APPL1 in this example), the mode name (0), and the local (resource manager's) fully qualified LU name (0). The locally known LU name in the connection pending extended data is sometimes called the "connect back" locally known LU name because it is used for connecting back to that LU. The mode name and local fully qualified LU name are both 0 because these parameters are not used when the partners are on the same processor.

   To connect to the CRR recovery server, the resource manager needs to determine the CRR recovery server's locally known LU name. Because the resource adapter and the CRR recovery server are always on the same processor, the resource manager can use the "connect back" locally known LU name from the connection pending extended data. The resource manager interprets the LU name qualifier *USERID to mean that the resource manager must use *IDENT 0 as the locally known LU name for connecting to that LU.

   The resource manager stores the locally known LU name, mode name, and local fully qualified LU name to use later. However, at this point the resource manager does *not* record any data in its log name log.

   After the resource manager accepts the connection, the resource adapter receives connection complete extended data. From this data, the resource adapter must save the local (resource adapter's) fully qualified LU name (0) and the remote (resource manager's) fully qualified name (0) to use in the CRR registration.

3. The resource manager receives the CRR recovery server's log name (CRR25) and TPN (.RS1) in the data buffer from the resource adapter and determines if an exchange of log names with the CRR recovery server is required. The resource manager looks for:

   a. A record in its log name log.

      The resource manager uses locally known LU name *IDENT 0, TPN .RS1, and log name CRR25 as search arguments. In this example, we are assuming that there is no matching record.

      Note: The resource manager must use the TPN as well as the locally known LU name to search the log. Several CRR recovery servers can exist in a TSAF collection and share a common LU out of the collection to the resource manager (through AVS).

      The CRR recovery server's locally known LU name and TPN are also used to determine the status of the resource manager's log if an initial Exchange-Log-Names message is sent. If both names match the log, the log is warm. If either name does not match the log or is missing, the log is cold.

   b. A local indication that log names were exchanged.

      This indication can be a local flag associated with each log name record, a local caching of the log name record, or some other method. Whatever technique is chosen, its purpose is to determine if the resource manager has exchanged log names with the CRR recovery server during the resource manager's current activation.

In cases where a resource manager could accidentally use the wrong log (as opposed to intentionally cold-logging), such as where logs are mounted or archived, it is important to force an exchange of log names with the CRR recovery server at least once for each activation of the resource manager to:

- Catch a warm/warm log mismatch, where it is possible that either the CRR recovery server's log or the resource manager's log is the wrong one.
- Avoid having to check for this warm/warm mismatch on every connect—just once each time the resource manager is activated.
- Accomplish an exchange of log names in case the CRR recovery server has erased log entries for the resource without cold-logging.

In our example, because there is no matching record in the resource manager's log and no local indication of a previous exchange, an exchange is required. The resource manager formulates an Exchange Log Names request that includes the following data:

Parameter	Value	Meaning
log status flag	X'00'	COLD—the CRR recovery server's locally known LU name and TPN were not known to the resource manager (not found in its log).
local fully qualified LU name	0	The resource manager has no local fully qualified LU name.
TPN	J	This is the resource manager's TPN.
log name 1	RESLOG5	This is the resource manager's log name.

The resource manager then connects to the CRR recovery server at locally known LU name *IDENT 0, TPN .RS1, mode name 0 and sends the Exchange Log Names message.

4. The CRR recovery server recognizes TPN .RS1 on the incoming connection as the special case of an Exchange Log Names request from a participating resource manager.

   In the connection from the resource manager, the CRR recovery server receives connection pending extended data (allocate data). This data includes the sender's (resource manager's) locally known LU name (*USERID RM1), the local (CRR recovery server's) fully qualified LU name (0), and the remote (resource manager's) fully qualified LU name (0). The CRR recovery server interprets the qualifier *USERID in the locally known LU name to mean that the CRR recovery server must use locally known LU name *IDENT 0 for connecting back to the resource manager. The fully qualified LU names are both 0 because these parameters are not used in a TSAF collection.

   The CRR recovery server receives the Exchange Log Names message, which contains the resource manager's TPN (J) and log name (RESLOG5). The CRR recovery server then looks for locally known LU name *IDENT 0, remote fully qualified LU name 0, TPN J, and log name RESLOG5 in its log. In our example, the CRR recovery server finds no match for this combination, so it adds a record, as follows:

Table 75. Example of a Log Name Table Record in the CRR Recovery Server's Log, TSAF Collection Case

LNLU	TPN	LOC FQLU	REM FQLU	LOGNAME
*IDENT 0	J	0	0	RESLOG5

5. The CRR recovery server formulates an Exchange Log Names reply that includes the following data and sends it to the resource manager:

Parameter	Value	Meaning
log status flag	X'00'	COLD—the resource manager's locally known LU name and TPN were not known to the CRR recovery server (not found in the log name table).

Parameter	Value	Meaning
local fully qualified LU name	0	The CRR recovery server has no local fully qualified LU name.
TPN	.RS1	This is the CRR recovery server's TPN.
log name 1	CRR25	This is the CRR recovery server's log name.

6. The resource manager receives the Exchange Log Names reply. Taking the CRR recovery server's log name and TPN from the message, together with the CRR recovery server's locally known LU name and the local (resource manager's) fully qualified LU name captured previously (Step 2), the resource manager adds a record to its log name log, as follows:

Table 76. Example of a Log Name Record in the Resource Manager's Log, TSAF Collection Case

LNLU	TPN	LOC FQLU	LOGNAME
*IDENT 0	.RS1	0	CRR25

Note:

a. Both LU names are important for a possible "shoulder tap", if the resource manager needs to notify the CRR recovery server of the resource manager's presence and readiness to accept resynchronization communications. The resource manager uses the locally known LU name and the TPN to connect to the CRR recovery server. The resource manager then sends its own fully qualified LU name and log name in an Exchange Log Names request.

b. There are cases where the local fully qualified LU name can change, such as moving from TSAF to AVS.

7. The resource adapter calls the CSL routine DMSREG to register with the SPM. Using this routine, the resource adapter provides the resource TPN (J), the resource recovery TPN (K), a recovery token, the mode name (0), the local fully qualified LU name (0), and the remote fully qualified LU name (0), along with other parameters.

   Note: The resource recovery TPN (K) is optional, and is assigned by the resource manager. In case of a failure during a sync point, the CRR recovery server uses the resource recovery TPN, if defined, to reach the resource manager. This lets the resource manager know that the connection is for resynchronization and not a normal data request. If a resource recovery TPN is not provided, the resource adapter must specify PIP data (also defined by the resource manager) in the registration. In that case, the CRR recovery server uses the resource TPN (J in this example) to reach the resource manager, but includes the PIP data on the connect to indicate that the connection is for resynchronization.

8. The SPM stores (for subsequent logging) the following information to represent the resource:
   - Resource TPN (J)
   - Local fully qualified LU name (0)
   - Remote fully qualified LU name (0)
   - Mode name (0)
   - Resource recovery TPN (K)
   - Recovery token
   - Other parameters not covered in this example.
9. At the start of a commit, the CRR recovery server receives a write log request from the SPM and writes an SPM Pending log record in the CRR recovery server's log. This log record contains the resource information listed in the previous step.
10. Assume that a failure occurs during a sync point for this application and that resynchronization recovery is started to recover this resource. The SPM Pending record for this resource in the CRR recovery server's log contains the following information:

Table 77. Example of an SPM Pending Record in the CRR Recovery Server's Log, TSAF Collection Case

LOC FQLU	REM FQLU	LUWID	TPN	REC TPN	REC TOKEN	MODENAME
0	0		J	K		0

Note: The LUWID and recovery token values are not shown only because they are not critical in this example, which is concentrating on LUs, TPNs, log names, and so on.

The CRR recovery server uses local fully qualified LU name 0, remote fully qualified LU name 0, and TPN J to search the log name table in its log for a matching record (see Table 75 on page 590). From that entry, the CRR recovery server gets the resource manager's locally known LU name, *IDENT 0.

Using information from the SPM Pending record and the log name table, the CRR recovery server connects to the resource manager at locally known LU name *IDENT 0, TPN K, mode name 0.

The CRR recovery server formulates an Exchange Log Names request that includes the following data and sends it to the resource manager:

Parameter	Value	Meaning
log status flag	X'01'	WARM—the CRR recovery server has a record in its log name table that contains the combination of local fully qualified LU name 0, remote fully qualified LU name 0, and TPN J. Note that the CRR recovery server's log is always warm in a resynchronization recovery Exchange Log Names request.
local fully qualified LU name	0	The CRR recovery server has no local fully qualified LU name.
TPN	.RS1	This is the CRR recovery server's TPN.
log name 1	CRR25	This is the CRR recovery server's log name.
log name 2	RESLOG5	This is the resource manager's log name that the CRR recovery server has saved in its log.

In the same buffer as the Exchange Log Names request, the CRR recovery server sends a Compare States request that identifies the LUWID and the current state of the logical unit of work.

11. The resource manager recognizes TPN K on the incoming connection as a special case, a resynchronization request from the CRR recovery server.

    The connection pending extended data (allocate data) that the resource manager receives contains the CRR recovery server's locally known LU name, *USERID RS1. The resource manager interprets this to mean that the resource manager must use locally known LU name *IDENT 0 when searching its log for a record for the CRR recovery server.

    After accepting the connection, the resource manager receives the Exchange Log Names message, which contains the CRR recovery server's TPN (.RS1) and log name (CRR25). Then the resource manager uses locally known LU name *IDENT 0 and TPN .RS1 to search for a matching log name record in its log. In our example, the resource manager finds such a record (see Table 76 on page 591) and verifies that the associated log name saved in the log (CRR25) matches the log name sent in the message. The resource manager can also verify that its own log name sent in the message (RESLOG5) is correct.

    The resource manager formulates an Exchange Log Names reply to confirm that everything matches. The reply includes the following data:

Parameter	Value	Meaning
log status flag	X'01'	WARM—the resource manager has previously communicated with the CRR recovery server.
local fully qualified LU name	0	The resource manager has no local fully qualified LU name.
TPN	K	This is the resource manager's recovery TPN.
log name 1	RESLOG5	This is the resource manager's log name.

The resource manager holds the Exchange Log Names reply and sends it to the CRR recovery server in the same buffer as the Compare States reply.

Note: The resolution of the Compare States request is not covered in this example.

12. The CRR recovery server processes the Exchange Log Names and Compare States replies and severs the connection to (deallocates the conversation with) the resource manager.

SNA Network Case

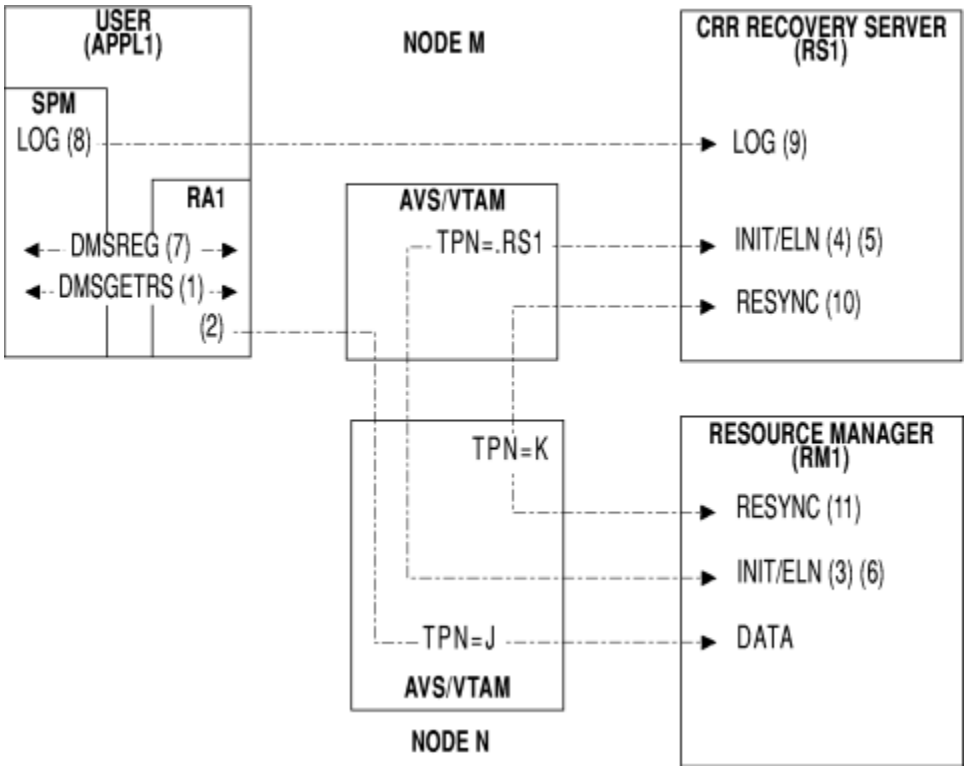


Figure 113. CRR Communications in an SNA Network

1. User virtual machine APPL1 is running a CMS application that uses the resource managed by resource manager RM1.

   Resource adapter RA1, RM1's adapter in the APPL1 virtual machine, calls the CSL routine DMSGETRS to get the CRR recovery server's current log name and TPN from the CRR sync point manager (SPM). In this example the values are CRR25 and .RS1, respectively.

2. Because CMS communications directory (COMDIR) resolved information is needed to register the resource for CRR, the resource adapter should issue an explicit resolve for the resource manager. In this example, assume that the resource manager's COMDIR name resolves to locally known LU name A B, TPN J, and mode name G.

The resource adapter then establishes a conversation with (connects to) the resource manager at locally known LU name A B, TPN J, mode name G and sends the CRR recovery server's log name and TPN in a data buffer. (The resource adapter sends this information in a data buffer each time it connects to the resource manager.)

When the resource adapter initiates the connection, the resource manager receives connection pending extended data (allocate data). Before the resource manager accepts the connection, it must get certain information from this data. (The connection pending extended data is not saved after the resource manager accepts the connection.)

Among the parameters in the connection pending extended data are the sender's (resource adapter's) locally known LU name (X Y in this example), the mode name (G), and the local (resource manager's) fully qualified LU name (N.X). The locally known LU name in the connection pending extended data is sometimes called the "connect back" locally known LU name because it is used for connecting back to that LU. Note that this locally known LU name is different from that used at the resource adapter's end of the link.

To connect to the CRR recovery server, the resource manager needs to determine the CRR recovery server's locally known LU name. Because the resource adapter and the CRR recovery server are always on the same processor, the resource manager uses the resource adapter's "connect back" locally known LU name from the connection pending extended data as the locally known LU name for connecting to the CRR recovery server.

The resource manager stores the locally known LU name (X Y), mode name (G), and local fully qualified LU name (N.X) to use later. However, at this point the resource manager does *not* record any data in its log name log.

After the resource manager accepts the connection, the resource adapter receives connection complete extended data. From this data, the resource adapter must save the local (resource adapter's) fully qualified LU name (M.A) and the remote (resource manager's) fully qualified LU name (N.X) to use in the CRR registration.

3. The resource manager receives the CRR recovery server's log name (CRR25) and TPN (.RS1) in the data buffer from the resource adapter and determines if an exchange of log names with the CRR recovery server is required. The resource manager looks for:

   a. A record in its log name log.

      The resource manager uses locally known LU name X Y, TPN .RS1, and log name CRR25 as search arguments. In this example, we are assuming that there is no matching record.

      Note: Because a resource manager could be accessed by various resource adapters on a particular processor through more than one LU, the resource manager's log could have multiple records containing the same CRR recovery server log name, but each with a different LU name.

      The resource manager also uses the CRR recovery server's locally known LU name and TPN to determine the status of the resource manager's log if an initial Exchange Log Names message is sent. If both names match the log, the log is warm. If either name does not match the log or is missing, the log is cold.

   b. A local indication that log names were exchanged.

      This indication can be a local flag associated with each log name record, a local caching of the log name record, or some other method. Whatever technique is chosen, its purpose is to determine if the resource manager has exchanged log names with the CRR recovery server during the resource manager's current activation.

      In cases where a resource manager could accidentally use the wrong log (as opposed to intentionally cold-logging), such as where logs are mounted or archived, it is important to force an exchange of log names with the CRR recovery server at least once for each activation of the resource manager to:

      - Catch a warm/warm log mismatch, where it is possible that either the CRR recovery server's log or the resource manager's log is the wrong one.

- Avoid having to check for this warm/warm mismatch on every connect—just once each time the resource manager is activated.
- Accomplish an exchange of log names in case the CRR recovery server has erased log entries for the resource without cold-logging.

In our example, because there is no matching record in the resource manager's log and no local indication of a previous exchange, an exchange is required. The resource manager formulates an Exchange Log Names request that includes the following data:

Parameter	Value	Meaning
log status flag	X'00'	COLD—the CRR recovery server's locally known LU name and TPN were not known to the resource manager (not found in its log).
local fully qualified LU name	N.X	This is the resource manager's local fully qualified LU name.
TPN	J	This is the resource manager's TPN.
log name 1	RESLOG5	This is the resource manager's log name.

The resource manager then connects to the CRR recovery server at locally known LU name X Y, TPN .RS1, mode name G and sends the Exchange Log Names message.

Note: The . prefix on the TPN signals the receiving AVS that the connection is to a global resource, even if a private gateway was used.

4. The CRR recovery server recognizes TPN .RS1 on the incoming connection as the special case of an Exchange Log Names request from a participating resource manager.

   In the connection from the resource manager, the CRR recovery server receives connection pending extended data (allocate data). From this data, the CRR recovery server gets the sender's (resource manager's) locally known LU name (A B), the local (CRR recovery server's) fully qualified LU name (M.A), and the remote (resource manager's) fully qualified LU name (N.X).

   The CRR recovery server receives the Exchange Log Names message, which contains the resource manager's TPN (J) and log name (RESLOG5). The CRR recovery server then looks for locally known LU name A B, remote fully qualified LU name N.X, TPN J, and log name RESLOG5 in its log name table. In our example, the CRR recovery server finds no match for this combination, so it adds a record, as follows:

Table 78. Example of a Log Name Table Record in the CRR Recovery Server's Log, SNA Network Case

LNLU	TPN	LOC FQLU	REM FQLU	LOGNAME
A B	J	M.A	N.X	RESLOG5

5. The CRR recovery server formulates an Exchange Log Names reply that includes the following data and sends it to the resource manager:

Parameter	Value	Meaning
log status flag	X'00'	COLD—the resource manager's locally known LU name and TPN were not known to the CRR recovery server (not found in the log name table).
local fully qualified LU name	M.A	This is the CRR recovery server's local fully qualified LU name.
TPN	.RS1	This is the CRR recovery server's TPN.
log name 1	CRR25	This is the CRR recovery server's log name.

6. The resource manager receives the Exchange Log Names reply. Taking the CRR recovery server's log name and TPN from the message, together with the CRR recovery server's locally known LU name

and the local (resource manager's) fully qualified LU name captured previously (Step 2), the resource manager adds a record to its log name log, as follows:

Table 79. Example of a Log Name Record in the Resource Manager's Log, SNA Network Case

LNLU	RS TPN	LOC FQLU	LOGNAME
X Y	.RS1	N.X	CRR25

Note:

a. Both LU names are important for a possible "shoulder tap", if the resource manager needs to notify the CRR recovery server of the resource manager's presence and readiness to accept resynchronization communications. The resource manager uses the locally known LU name and the TPN to connect to the CRR recovery server. The resource manager then sends its own fully qualified LU name and log name in an Exchange Log Names request.

b. There are cases where the local fully qualified LU name can change, such as moving from TSAF to AVS. In the case where there are multiple paths to a target CRR recovery server from the resource manager, there can be more than one local fully qualified LU name.

7. The resource adapter calls the CSL routine DMSREG to register with the SPM. Using this routine, the resource adapter provides the resource TPN (J), the resource recovery TPN (K), a recovery token, the mode name (G), the local fully qualified LU name (M.A), and the remote fully qualified LU name (N.X), along with other parameters.

   Note: The resource recovery TPN (K) is optional, and is assigned by the resource manager. In case of a failure during a sync point, the CRR recovery server uses the resource recovery TPN, if defined, to reach the resource manager. This lets the resource manager know that the connection is for resynchronization and not a normal data request. If a resource recovery TPN is not provided, the resource adapter must specify PIP data (also defined by the resource manager) in the registration. In that case, the CRR recovery server uses the resource TPN (J in this example) to reach the resource manager, but includes the PIP data on the connect to indicate that the connection is for resynchronization.

8. The SPM stores (for subsequent logging) the following information to represent the resource:
   - Resource TPN (J)
   - Local fully qualified LU name (M.A)
   - Remote fully qualified LU name (N.X)
   - Mode name (G)
   - Resource recovery TPN (K)
   - Recovery token
   - Other parameters not covered in this example.
9. At the start of a commit, the CRR recovery server receives a write log request from the SPM and writes an SPM Pending log record in the CRR recovery server's log. This log record contains the resource information listed in the previous step.
10. Assume that a failure occurs during a sync point for this application and that resynchronization recovery is started to recover this resource. The SPM Pending record for this resource in the CRR recovery server's log contains the following information:

Table 80. Example of an SPM Pending Record in the CRR Recovery Server's Log, SNA Network Case

LOC FQLU	REM FQLU	LUWID	TPN	REC TPN	REC TOKEN	MODENAME
M.A	N.X		J	K		G

Note: The LUWID and recovery token values are not shown only because they are not critical in this example, which is concentrating on LUs, TPNs, log names, and so on.

The CRR recovery server uses local fully qualified LU name M.A, remote fully qualified LU name N.X, and TPN J to search the log name table in its log for a matching record (see Table 78 on page 595). From that entry, the CRR recovery server gets the resource manager's locally known LU name, A B.

Using information from the SPM Pending record and the log name table, the CRR recovery server connects to the resource manager at locally known LU name A B, TPN K, mode name G.

The CRR recovery server formulates an Exchange Log Names request that includes the following data and sends it to the resource manager:

Parameter	Value	Meaning
log status flag	X'01'	WARM—the CRR recovery server has a record in its log name table that contains the combination of local fully qualified LU name M.A, remote fully qualified LU name N.X, and TPN J. Note that the CRR recovery server's log is always warm in a resynchronization recovery Exchange Log Names request.
local fully qualified LU name	M.A	This is the CRR recovery server's local fully qualified LU name.
TPN	.RS1	This is the CRR recovery server's TPN.
log name 1	CRR25	This is the CRR recovery server's log name.
log name 2	RESLOG5	This is the resource manager's log name that the CRR recovery server has saved in its log.

In the same buffer as the Exchange Log Names request, the CRR recovery server sends a Compare States request that identifies the LUWID and the current state of the logical unit of work.

11. The resource manager recognizes TPN K on the incoming connect as a special case, a resynchronization request from the CRR recovery server.

    From the connection pending extended data, the resource manager gets the CRR recovery server's locally known LU name (X Y).

    After accepting the connection, the resource manager receives the Exchange Log Names message, which contains the CRR recovery server's TPN (.RS1) and log name (CRR25). The resource manager then uses locally known LU name X Y and TPN .RS1 to search for a matching log name record in its log. In our example, the resource manager finds such a record (see Table 78 on page 595) and verifies that the associated log name saved in the log (CRR25) matches the log name sent in the message. The resource manager can also verify that its own log name sent in the message (RESLOG5) is correct.

    The resource manager formulates an Exchange Log Names reply to confirm that everything matches. The reply includes the following data:

Parameter	Value	Meaning
log status flag	X'01'	WARM—the resource manager has previously communicated with the CRR recovery server.
local fully qualified LU name	N.X	This is the resource manager's local fully qualified LU name.
TPN	K	This is the resource manager's recovery TPN.
log name 1	RESLOG5	This is the resource manager's log name.

    The resource manager holds the Exchange Log Names reply and sends it to the CRR recovery server in the same buffer as the Compare States reply.

Note: The resolution of the Compare States request is not covered in this example.

12. The CRR recovery server processes the Exchange Log Names and Compare States replies and severs the connection to (deallocates the conversation with) the resource manager.

Appendix J. ISPF Example

Using ISPF, the following FORTRAN program, FORT2, lets the user add, change, delete, or display records in a file of peoples' names, by serial number. Records must be added before they can be changed, deleted or displayed. (This application does the same things as the program in “Example 2: Complete FORTRAN Application” on page 537.)

```
      IMPLICIT INTEGER (A-Z)
      CHARACTER*1  FNTYPE,FBLNK
      CHARACTER*6  EMPSER,EMPBLK
      CHARACTER*16 FNAME,LNAME,NAMEBL
      DATA  NAMEBL /'                '/
      DATA  EMPBLK /'      '/
      DATA  FBLNK /' '/
      LASTRC = ISPLNK ('VDEFINE','(FNAME)',FNAME,'CHAR',16)
      LASTRC = ISPLNK ('VDEFINE','(LNAME)',LNAME,'CHAR',16)
      LASTRC = ISPLNK ('VDEFINE','(EMPSER)',EMPSER,'CHAR',6)
      LASTRC = ISPLNK ('VDEFINE','(F)',FNTYPE,'CHAR',1)
      LASTRC = ISPLNK ('TBOPEN','EMPLTBL ')
      IF (LASTRC.EQ.0) GO TO 10
      LASTRC = ISPLNK ('TBCREATE','EMPLTBL ','(EMPSER)',
     * '(LNAME FNAME)')
10    FNTYPE = FBLNK
      EMPSER = EMPBLK
      LASTRC = ISPLNK ('DISPLAY','MENUPAN ')
      IF (LASTRC.EQ.8) GO TO 70
      IF (FNTYPE.GT.'4') GO TO 70
      LASTRC = ISPLNK ('TBGET','EMPLTBL ')
      IF (FNTYPE.EQ.'1'.AND.LASTRC.NE.0) GO TO 20
      IF (FNTYPE.GT.'1'.AND.LASTRC.EQ.0) GO TO 30
      LASTRC = ISPLNK ('SETMSG','MSG002  ')
      GO TO 10
20    FNAME = NAMEBL
      LNAME = NAMEBL
30    LASTRC = ISPLNK ('SETMSG','MSG001  ')
      IF (FNTYPE.EQ.'3') GO TO 40
      LASTRC = ISPLNK ('DISPLAY','NAMEPAN ')
      IF (FNTYPE.EQ.'1') GO TO 50
      IF (FNTYPE.EQ.'2') GO TO 60
      GO TO 10
40    LASTRC = ISPLNK ('TBDELETE','EMPLTBL ')
      GO TO 10
50    LASTRC = ISPLNK ('TBADD','EMPLTBL ')
      GO TO 10
60    LASTRC = ISPLNK ('TBPUT','EMPLTBL ')
      GO TO 10
70    CONTINUE
      LASTRC = ISPLNK ('TBCLOSE','EMPLTBL ')
      LASTRC = ISPLNK ('VRESET  ')
      STOP
      END
```

Figure 114. ISPF Example

The ISPF specification for the MENUPAN panel is:

```
)BODY
%------------------------------SELECTION-----------------------
%COMMAND ===>_ZCMD                                             +
+
+
%SELECT REQUIRED FUNCTION AND ENTER SERIAL NUMBER BELOW
+
+
+   1 - ADD, 2 - CHANGE, 3 - ERASE, 4 - DISPLAY, 5 - END
+
+
+  FUNCTION NUMBER%===>_FNTYPE+
+
+
+    SERIAL NUMBER%===>_EMPSER+      (MUST BE 6 NUMERIC DIGITS)
+
+
```

```
+
)INIT
  .CURSOR = F
)PROC
  VER (&F, NONBLANK)
  VER (&F, PICT,N)
  IF (&F &notsym.=5)
    VER (&EMPSER, NONBLANK)
    VER (&EMPSER, PICT,NNNNNN)
)END
```

This is the way it is displayed on the screen:

```
------------------------------SELECTION-----------------------
COMMAND ===>


SELECT REQUIRED FUNCTION AND ENTER SERIAL NUMBER BELOW


    1 - ADD, 2 - CHANGE, 3 - ERASE, 4 - DISPLAY, 5 - END


   FUNCTION NUMBER ===>


     SERIAL NUMBER ===>              (MUST BE 6 NUMERIC DIGITS)
```

The ISPF specification for the NAMEPAN panel is:

```
)BODY
%----------------------------NAME PANEL--------------------------
+
+
+
+           SERIAL NUMBER%===>_EMPSER+
+
+
+       FIRST NAME%===>_FNAME             +
+
+
+       LAST NAME%===>_LNAME              +
+
+
+
)PROC
  .CURSOR = FNAME
  VER (&FNAME,ALPHA)
  VER (&LNAME,ALPHA)
)END
```

This is the way it is displayed on the screen:

```
----------------------------NAME PANEL-------------------------


            SERIAL NUMBER ===>

         FIRST NAME ===>

         LAST NAME ===>
```

The ISPF specifications of the two messages issued by FORT2 are:

```
MSG001   'OPERATION COMPLETED'                    .ALARM=NO
'THE OPERATION SPECIFIED HAS BEEN COMPLETED.'
MSG002   'INVALID OPERATION'                      .ALARM=YES
'ENTER A NUMBER FROM 1 TO 5 IN THE SPACE PROVIDED.'
```

You should issue the following FILEDEF commands before running the example program, FORT2:

```
FILEDEF ISPPROF DISK ISPPROF MACLIB A (PERM
FILEDEF ISPPLIB DISK USERPAN MACLIB * (PERM CONCAT
FILEDEF ISPPLIB DISK ISRPLIB MACLIB * (PERM CONCAT
FILEDEF ISPPLIB DISK ISPPLIB MACLIB * (PERM CONCAT
FILEDEF ISPMLIB DISK EXAMMSG MACLIB * (PERM CONCAT
FILEDEF ISPMLIB DISK ISRMLIB MACLIB * (PERM CONCAT
FILEDEF ISPMLIB DISK ISPMLIB MACLIB * (PERM CONCAT
FILEDEF ISPSLIB DISK ISRSLIB MACLIB * (PERM CONCAT
FILEDEF ISPTABL DISK MYTABLE MACLIB A (PERM
FILEDEF ISPTLIB DISK MYTABLE MACLIB * (PERM CONCAT
FILEDEF ISPTLIB DISK ISRTLIB MACLIB * (PERM CONCAT
FILEDEF ISPTLIB DISK ISPTLIB MACLIB * (PERM CONCAT
FILEDEF ISPXLIB DISK VFORTLIB TXTLIB * (PERM CONCAT
FILEDEF ISPXLIB DISK MYLIB    TXTLIB * (PERM CONCAT
```

Appendix K. MQ Series Applications

The following material shows the steps and files needed to create MQ Series applications. The information is divided according to the language used in the application.

The MQ Client code is shipped with z/VM and is part of the CMS libraries.

For information on loading TXTLIBs, LOADLIBs, and MACLIBs, see Chapter 6, "Loading and Running Your Program," on page 47. For information on CMS libraries see Chapter 19, "Creating and Manipulating the CMS Libraries," on page 305.

Other TXTLIBs and MACLIBs may be required depending on the compiler and transaction protocols being used.

See *MQSeries Application Programming Guide,* SC33–0807, for more information on MQ Series sample files.

C Applications

Use the information below to compile, install, and test MQ Series applications written in C.

The following MQ Series Client code libraries are required for C:

```
AMQTEXT   TXTLIB
AMQTEXTC  TXTLIB
CMQC   H
```

C Sample Files

Sample files:

- AMQSPUT0 C: Sample program to test MQPUT
- AMQSGET0 C: Sample program to test MQGET
- AMQSBCG0 C: Sample program to read and output both the message descriptor fields and the message content of all the messages on a queue
- AMSQSGBR0 C: Sample C program that displays messages on a message queue (example using Browse option of MQGET)
- AMQSECHA C: Sample C program — echo messages to reply to queue
- AMQSINQ2 C: Sample program to test MQINQ
- AMQSSET2 C: Sample program to test MQSET

C Sample

Use the following steps to create a sample MQ Series application with the AMQSPUT0 sample file:

Access the libraries:

MQSeries client code
LE/370 libraries
C libraries
TCP/IP or APPC libraries

Identify the libraries to be searched:
The MQ VM Client code is in the AMQTEXT TXTLIB. The C programming language interface is in the AMQTEXTC TXTLIB. The LE/370 code is in the SCEELKED TXTLIB and SCEERUN LOADLIB. CMQC H

contains the MQI call and structure prototypes needed to compile C programs using MQSeries MQI calls.

global txtlib sceelked amqtext amqtextc commtxt cmssaa
global loadlib sceerun

Compile the C Application:
The TEXT deck will be generated on your A-disk.

Build the Application:

1. Link the TEXT file:

   CPLINK AMQSPUT0

   This generates a fi le named CPOBJ TEXT on your A-disk.
2. Load the object:

   LOAD CPOBJ
3. Generate the module:

   GENMOD AMQSPUT0 (from CEESTART on A-disk)
   AMQSPUT0 module (the load module)
   Load map

Execute the application

Access the libraries.

Define the MQSERVER variable in LASTING GLOBALV.

globalv select cenv setlp mqserver system.def.svrconn/tcp/xx.xx.xx.x

Then execute:

AMQSPUT0 QUEUE1 qmgr

where qmgr is queue manager on the server

The following message is displayed:

Sample AMQSPUT0 start
target name is QUEUE1

Type some message and press Enter twice. The following message is displayed:

Sample AMQSPUT0 end

Message is on the queue.

COBOL and PL/I Applications

The following libraries are required:

AMQTEXT TXTLIB
AMQTEXTL TXTLIB
AMQOM MACLIB

COBOL Sample Files

AMQ0PUT0 COBOL
Sample program to test MQPUT

AMQ0GET0 COBOL
Sample program to test MQGET

PL/I Sample Files

AMQPBRW PLI
Sample program to browse message on a queue.

AMQPGET PLI
Sample program to test MQGET.

AMQPPUT PLI
Sample program to test MQPUT.

COBOL and PL/I Samples

Use the following steps to create a sample MQ Series application with the AMQ0PUT0 file:

Access the libraries:

MQSeries client code
LE/370 libraries
TCP/IP or APPC libraries

Identify the libraries to be searched:
The MQ VM Client code is in the AMQTEXT TXTLIB. The COBOL and PL/I programming interfaces are in the AMQTEXTL TXTLIB. The LE/370 code is in the SCEELKED TXTLIB and SCEERUN LOADLIB. The AMQOM MACLIB contains the COPY files required to build and execute COBOL and PL/I applications using MQ Series MQI calls.

global txtlib sceelked amqtext amqtextl commtxt cmssaa
global loadlib sceerun
global maclib amqom ...

Compile the COBOL application:
The TEXT deck will be generated on your A-disk.

Build the application:

1. Link the TEXT file:

   CPLINK AMQ0SPUT0

   This generates a file CPOBJ TEXT on your A-disk.
2. Load the object.

   LOAD CPOBJ
3. Generate the module:

   For COBOL:

   GENMOD AMQ0PUT0

   For PL/I:

   GENMOD AMQPPUT (from PLISTART)

Execute the application

Access the libraries.

Define the MQSERVER variable in LASTING GLOBALV:

```
globalv select cenv setlp mqserver system.def.svrconn/tcp/xx.xx.xx.x
AMQ0PUT0 QUEUE1 qmgr
where qmgr is queue manager on the server
```

The following message is displayed:

```
Sample AMQSPUT0 start
target name is QUEUE1
```

Type some message and press Enter twice. The following message is displayed:

```
Sample AMQSPUT0 end
```

Message is on the queue.

Assembler Applications

This section describes how to build an MQ Series Assembler routine. Assembler sample files are not shipped with z/VM. See *MQSeries Application Programming Guide,* SC33–0807, for information on MQ Series Assembler sample files.

Use the information below to compile, install, and test MQSeries applications written in Assembler.

The following libraries are required:

```
AMQTEXT   TXTLIB
AMQTEXTA TXTLIB
AMQOM  MACLIB
```

Assembler Sample

Use the following steps to create your sample MQ Series application using AMQASPUT as your sample file:

Access the libraries:

MQSeries Client code
LE/370 libraries
TCP/IP or APPC libraries

Identify the libraries to be searched:

The MQ VM Client code is in the AMQTEXT TXTLIB. The Assembler programming language interface is in the AMQTEXTA TXTLIB. The LE/370 code is in the SCEELKED TXTLIB and SCEERUN LOADLIB. The AMQOM MACLIB contains the COPY files required to build and execute Assembler applications using MQ Series MQI calls.

```
global txtlib sceelked amqtext amqtexta commtxt cmssaa
global loadlib sceerun
global maclib amqom ...
```

Compile the application:

The TEXT deck will be generated on your A-disk.

Build the application

1. Link the TEXT file:

   ```
   CPLINK AMQASPUT
   ```

   This generates a file CPOBJ TEXT on your A-disk.
2. Load the object.

   ```
   LOAD CPOBJ
   ```
3. Generate the module:

GENMOD AMQASPUT
on A-disk.
AMQASPUT module (the LOAD module)
LOAD MAP

Execute the application

Access the libraries.

Define the MQSERVER variable in LASTING GLOBALV:

globalv select cenv setlp mqserver system.def.svrconn/tcp/xx.xx.xx.x
AMQASPUT QUEUE1 qmgr
where qmgr is queue manager on the server

REXX Applications

Use the information below to compile, install, and test MQ Series applications written in REXX.

The following library is required:

AMQLLIB LOADLIB

The RXMQV module is the REXX interface module that maps the REXX MQI calls into the MQSeries for Client libraries. You must NUCXLOAD this module.

REXX Sample Files

RXMQVPUT EXEC
Sample EXEC to put messages to a queue

RXMQVGET EXEC
Sample EXEC to get messages from a queue

REXX Sample

Use the following steps to create a sample MQ Series application with the RXMQVPUT EXEC:

Access the libraries:

MQ Series Client code
LE/370 libraries
TCP/IP or APPC libraries

Identify the libraries to be searched:
The MQ VM Client code is on in the AMQLLIB LOADLIB. The LE/370 code is in the SCEELKED TXTLIB and SCEERUN LOADLIB. The RXMQV module is the REXX interface module that maps the REXX MQI calls into the MQSeries for Client libraries. You must NUCXLOAD this module.

global loadlib sceerun amqllib

Execute the application

Issue the following CMS command (this will keep C malloc storage around):

SET STORECLR ENDCMD

Access the libraries.

Define the MQSERVER variable in LASTING GLOBALV:

globalv select cenv setlp mqserver system.def.svrconn/tcp/xx.xx.xx.x

1. NUCXLOAD RXMQV (SYSTEM
2. RXMQVPUT QUEUE1 qmgr

where qmgr is queue manager on the server.

Appendix L. Data Compression Services

This appendix contains the following to assist you in compressing and expanding your data:

- Example of a dictionary build
- Using CSRCMPEV to test compression and expansion.

A Dictionary Build Using CSRBDICV

The following is an example of the message output of CSRBDICV that is written on your screen when the *msglevel* specification is 3. The spec file is TEXT SPECFILE, except with maxnodes changed from 40000 to 22000 in order to have approximately the same number of nodes built in each of the three program passes. Comments that explain the messages and some elements of the operation of the exec are placed after the corresponding messages. They are indented and highlighted.

CSRBDICV was invoked by entering: `bdict 4 1 eb ch5 script(text`

```
15 May 1993 17:58:19
BDICT dsize=4 sdfmt=1 erase=EB scanfile=CH5 SCRIPT A specfile=TEXT SPECFILE A
   Shows how the input command was parsed.
BDICT (C) Copyright IBM Corporation 1993
CH5 CEDICT41 A erased
CH5 ACDICT41 A erased
CH5 AEDICT41 A erased
CH5 BDICT41 A erased
   The files existed from a previous run.
Contents of specfile (TEXT SPECFILE A) are:

**The following is with a 4K-entry dictionary.
**Provides 30.88% compression (output/input) for the source of
**Chapter 5 of the ESA/390 Principles of Operation (30.32% if all output
**bits are concatenated together).
**Optimization (change x under opt to opt) improves compression by 0.7%.
**results maxnodes maxlevels msglevel stepping prperiod dicts
  r       22000    60        3        f 7 2 7  1000     af asm
**colaps opt treedisp treehex treenode dupccs
  aam     x   x        h       n        x
**FLD col type dcenmen              INT  intspec
  FLD 1   sa   dce 4                INT  aeis 1 (40)
                                    INT  a12b3s (40)
  FLD end
**Note: Some text will be compressed better if the INT aeis 1 (40) is
**omitted; i.e., try it with and without the INT aeis 1 (40).  Also, if
**the text is ASCII instead of EBCDIC, the 40 should be changed to 20.

Starting program pass 1
Starting pass 1 through file in program pass 1.  Stepping=F 7 2 7
Process line 1 of every 7 lines in pass 1.1
Line=456 node=1000 tree depth=8
   A progress report is issued after the building of each 1000
   nodes.  The report identifies the line currently being processed and
   the number of levels in the tree so far.
Line=1331 node=2000 tree depth=13
Line=2381 node=3000 tree depth=13
Line=3634 node=4000 tree depth=15
Line=5188 node=5000 tree depth=20
Line=7036 node=6000 tree depth=20
Line=8786 node=7000 tree depth=27
Lines in file <8807
   Gives the approximate size of the file (to within seven
   because of the stepping specification).
Starting pass 2 through file in program pass 1.  Stepping=F 7 2 7
Process line 3 of every 7 lines in pass 1.2
Line=605 node=7339 tree depth=27 at end of program pass 1
Number of nodes at each level:
256 950 1772 1488 1030    642 414 317 189 114 58 41 26 13 6    4 4 3 1 1 2 2 1 1 1
1 2
Eliminate nodes by collapsing children into parents
Number of nodes eliminated at each level:
0 0 191 409 354    266 156 109 73 43 22 14 7 6 0    1 0 0 0 0 0 0 0 0 0
0 0
```

```
Nodes eliminated by collapsing= 1651
Total nodes eliminated= 0 + 1651 = 1651
Remaining nodes= 7339 - 1651 = 5688
Sort siblings by node values
   The value of a node is the number of times it has been
   encountered during scanning.  The children of a parent are placed
   in the order of their frequency of use.
Nodes counted during sorting= 5688
   The equality of this number to the previous 5688 evidences
   correct operation.
Eliminate nodes designated by duplicate CCs and SCs
   except for nodes designated by consecutive CCs beginning with first CC.
      Duplicate CCs and SCs can have resulted in this program
      pass from the building of double-character nodes.  They can be
      formed in the next pass because of the collapsing of children into
      parents in this pass.  For example, if parent A has a child with
      extension characters BC when the string ABD is to be matched,
      parent A must acquire a second child whose first extension
      character is B (since there cannot be a match on the first child).
      Then parent A will contain two CCs both of value B.
   Eliminate duplicate duplicate-CC nodes.
      These can result from moving an extension character from
      a parent to its child.
   For a duplicate-CC node whose initial extension characters equal all
   extension characters of a preceding node, interchange the nodes if the
   second is of equal or greater value, or else delete the second node.
      A subsequent node cannot be of greater value.
Total nodes eliminated because of duplicate CCs = 4, dup CCs= 32
   Dup. nodes eliminated= 0 (dup. nodes caused by use of colaps=AM or AAM)
   Nodes interchanged to avoid architecture violation= 0
      A duplicate character found in higher level nodes.
   Nodes eliminated to avoid architecture violation= 1
      This is the same violation.
Total nodes eliminated= 1651 + 4 = 1655
Remaining nodes= 7339 - 1655 = 5684
Nodes= 5684 Dict size= 4K Average value= 5.81
Trim tree to no more than 8K entries
   Only nodes with the cutoff value or more are kept.  Cutoff
   is explained later.  Trimming the tree causes new paths (strings)
   in the tree to result from interruptions (last possible match)
   after frequently used existing paths instead of infrequently used
   ones.  A smaller tree also provides faster scanning (matching).
Nodes remaining after program pass 1 = 5684
Total nodes eliminated after program pass 1 = 7339 - 5684 = 1655
Starting program pass 2
   The deletion of duplicate-CC and SC nodes at the end of
   pass 1 will cause new paths to be formed during this pass.
Line=1928 node=8000 tree depth=30
Line=4567 node=9000 tree depth=30
Line=7570 node=10000 tree depth=31
Starting pass 3 through file in program pass 2.  Stepping=F 7 2 7
Process line 5 of every 7 lines in pass 2.3
Line=1272 node=11000 tree depth=38
Line=4373 node=12000 tree depth=38
Line=8153 node=13000 tree depth=38
Starting pass 4 through file in program pass 2.  Stepping=F 7 2 7
Process line 7 of every 7 lines in pass 2.4
Line=2198 node=14000 tree depth=39
Line=4977 node=14668 tree depth=39 at end of program pass 2
Number of nodes at each level:
256 1320 2935 2897 2131    1412 866 489 286 169 84 48 29 16 10
10 9 5 5 6 5 4 3 2 1
1 2 1 1 2 2 1 1 1 2    1 2 1 1
Eliminate nodes by collapsing children into parents
Number of nodes eliminated at each level:
0 0 179 429 478    369 265 131 100 50 31 22 7 7 1    3 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0    0 0 0 0
Nodes eliminated by collapsing= 2073
Total nodes eliminated= 1655 + 2073 = 3728
Remaining nodes= 14668 - 3728 = 10940
Sort siblings by node values
Nodes counted during sorting= 10940
Eliminate nodes designated by duplicate CCs and SCs
   The explanatory messages are not repeated.
Total nodes eliminated because of duplicate CCs = 542, dup CCs= 247
   Dup. nodes eliminated= 0 (dup. nodes caused by use of colaps=AM or AAM)
   Nodes interchanged to avoid architecture violation= 0
   Nodes eliminated to avoid architecture violation= 21
Total nodes eliminated= 3728 + 542 = 4270
Remaining nodes= 14668 - 4270 = 10398
Nodes= 10398 Dict size= 4K Average value= 9.39
Trim tree to no more than 8K entries
```

```
Nodes remaining after program pass 2 = 8154
Total nodes eliminated after program pass 2 = 14668 - 8154 = 6514
Starting program pass 3
Line=6335 node=15000 tree depth=39
Starting pass 5 through file in program pass 3.  Stepping=F 7 2 7
Process line 2 of every 7 lines in pass 3.5
Line=1157 node=16000 tree depth=43
Line=4965 node=17000 tree depth=43
Starting pass 6 through file in program pass 3.  Stepping=F 7 2 7
Process line 4 of every 7 lines in pass 3.6
Line=375 node=18000 tree depth=43
Line=5597 node=19000 tree depth=45
Starting pass 7 through file in program pass 3.  Stepping=F 7 2 7
Process line 6 of every 7 lines in pass 3.7
Line=1315 node=20000 tree depth=47
Line=6768 node=21000 tree depth=47
Line=8805 node=21477 tree depth=47 at end of program pass 3
Number of nodes at each level:
256 1688 3888 3959 2919    1990 1312 724 419 225 106 60 41 28 23
16 13 10 5 8 8 6 4 4 5
3 3 2 1 2 2 1 1 2 2    1 2 1 1 1 1 1 1 1 1    1 1
Eliminate nodes by collapsing children into parents
   and possibly move extension chars from parent to child
Number of nodes eliminated at each level:
0 0 275 782 842    679 433 303 180 104 53 27 25 16 10    6 3 6 0 0 2 2 2 0 1
0 1 2 0 0 0 1 0 0 0    0 0 0 1 1 0 1 0 1 1    0 1
Nodes eliminated by collapsing= 3761
Total nodes eliminated= 6514 + 3761 = 10275
Remaining nodes= 21477 - 10275 = 11202
Sort siblings by node values
Nodes counted during sorting= 11202
Eliminate nodes designated by duplicate CCs and SCs
Total nodes eliminated because of duplicate CCs = 765, dup CCs= 455
   Dup. nodes eliminated= 7 (dup. nodes caused by use of colaps=AM or AAM)
   Nodes interchanged to avoid architecture violation= 0
   Nodes eliminated to avoid architecture violation= 26
Total nodes eliminated= 10275 + 765 = 11040
Remaining nodes= 21477 - 11040 = 10437
Nodes= 10437 Dict size= 4K Average value= 11.72
Cutoff value=2 Spares=?
   Cutoff is calculated by means of a formula.
Cutoff increment=7
   The increment is arbitrarily set to 7.
Cutoff value=2 Spares=-3637
   Spares is the number of entries that would be left in the
   dictionary if all nodes having the cutoff value or more were placed in
   the dictionary.  Because spares is negative, cutoff 2 is too low.
Cutoff value=9 Spares=718
Cutoff increment=1
   Because spares became positive, the increment is reduced to 1,
   and cutoff is set to the midpoint between the one that failed and the
   one that succeeded.
Cutoff value=5 Spares=-754
Cutoff value=6 Spares=-266
Cutoff value=7 Spares=156
101 sibling descriptors required
   The number of sibling descriptors required by the nodes with
   value 7 or more is determined.  If this exceeded spares, cutoff would
   be incremented by one.
Oldcutoff=6
   Spares is now 156 - 101 = 55.  This number of nodes with value
   6 will be placed in the dictionary.  This may require additional
   sibling descriptors.
Place child counts in nodes
   For each entry to be placed in the dictionary, it is determined
   how many children of the entry will be placed in the dictionary.
Nodes remaining after program pass 3 = 3995
   This is the number of character entries to be placed in the
   dictionary.
Total nodes eliminated after program pass 3 = 21477 - 3995 = 17482
Old cutoff nodes rejected because parent is also an old cutoff node= 9
   The node is rejected because there might be no spares left when
   the parent of the node was to be placed in the dictionary.  A parent
   and its child can have the same value only as a result of moving an
   extension character from the parent to the child or when
   optimization is used.
Old cutoff nodes rejected because only one node would go in a new SD= 1
   The node is rejected because to place it in the dictionary would
   consume two entries (one for a new sibling descriptor).
Total oldcutoff nodes rejected or ignored= 367
Oldcutoff-1 nodes selected= 0
   In rare cases, the number of oldcutoff nodes is less than spares,
```

```
   in which case nodes with value oldcutoff-1 are placed in the dictionary.
Prune the tree.
   All the work has already been done.
Numbers of unpruned nodes at each level were:
256 1354 2771 2501 1808    1080 674 338 164 87 40 24 15 9 9    7 8 5 8 6 5 3 4 3 2
2 2 1 2 2 1 2 2 1 2    1 1 1 1
Numbers of pruned nodes at each level are:
256 595 878 796 585    395 204 107 66 31 15 7 6 4 4    5 5 4 6 3 2 3 2 1 1
1 1 1 1 1 1 1 1 2    1 1 1 0
Pruned nodes divided by unpruned nodes as percentages are:
100% 43% 31% 31% 32% 36% 30% 31% 40% 35% 37% 29% 40% 44% 44% 71% 62% 80%
75% 50% 40% 100% 50% 33% 50%
   Describes only the first 25 levels.
Dictionary entries including sibling descriptors=4096 256 3739 101
   The second number is the number of alphabet entries, the third
   is the number of nonalphabet character entries, and the fourth is the
   number of sibling descriptors.
Test nodes=3739
   A test for correctness.
101 sibling descriptors required
   A test for correctness.
Write dictionary file CH5 CEDICT41 A
Entry 256 Number of SDs with counts 0-15 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
   Entry 256 is written first because the children of a parent must
   be written before the parent so a pointer to the children can be
   placed in the parent.  Except for the alphabet entries, there is no
   resemblance between the entry numbers in the dictionary and the node
   numbers in the tree.
Entry 512 Number of SDs with counts 0-15 0 1 0 0 1 0 0 0 0 0 1 0 0 0 0 0
   Describes the sibling descriptors that have been written so far.
Entry 768 Number of SDs with counts 0-15 0 1 2 0 2 0 2 0 1 0 1 0 0 0 1 1
   An SD with a count of 15 contains 14 SCs and indicates another SD.
Entry 1024 Number of SDs with counts 0-15 0 4 2 2 2 0 2 0 1 2 1 0 0 0 1 1
Entry 1280 Number of SDs with counts 0-15 0 6 2 3 3 0 2 0 1 2 1 0 1 0 1 1
Entry 1536 Number of SDs with counts 0-15 0 8 3 4 3 0 2 1 2 2 1 0 1 0 1 1
Entry 1792 Number of SDs with counts 0-15 0 12 3 4 5 0 2 1 2 2 1 0 1 0 1 2
Entry 2048 Number of SDs with counts 0-15 0 16 4 4 6 0 3 1 2 2 1 0 1 0 1 2
Entry 2304 Number of SDs with counts 0-15 0 19 5 4 8 0 4 1 2 2 1 1 1 0 1 2
Entry 2560 Number of SDs with counts 0-15 0 21 8 5 8 0 5 1 3 2 1 1 1 1 1 2
Entry 2816 Number of SDs with counts 0-15 0 22 9 5 8 0 5 1 4 2 2 1 1 1 1 2
Entry 3072 Number of SDs with counts 0-15 0 26 9 5 8 1 5 1 5 2 2 1 1 1 1 2
Entry 3328 Number of SDs with counts 0-15 0 29 10 6 9 1 5 2 5 2 4 1 1 1 1 2
Entry 3584 Number of SDs with counts 0-15 0 31 11 6 10 1 5 2 5 2 4 1 1 1 2 2
Entry 3840 Number of SDs with counts 0-15 0 32 14 8 11 1 5 4 5 2 4 1 1 1 2 2
Entry 4095 Number of SDs with counts 0-15 0 35 14 9 11 1 6 5 6 2 4 1 1 1 2 3
Total characters in dictionary=6897
Characters per entry= 6897/4K =  1.68
0 unused entry(s)
   In rare cases, this might be one.  If it is more than one, there
   is an error unless the scan file was too small to fill the dictionary.
Entry 0
   Entries 0-255 are written last.
Reading in dictionary
   The variables used to form the dictionary are dropped (storage is
   released), and a new set of variables is created.
Build ACDICT41
Write ACDICT41 and AEDICT41 but not BDICT41
   The dicts specification is af, not afd.
Done
```

Using CSRCMPEV to Test Compression and Expansion

The following is an example of the message output of CSRCMPEV that is written on your screen when using the *msglevel* argument of 1. Comments that explain the messages are placed after the corresponding messages. They are indented and highlighted.

CSRCMPEV was invoked by entering `csrcmpev 4 1 nhd ch5 script`

```
15 May 1993 17:59:58
CMPEXP dsize=4 sdfmt=1 expnd=NHD scanfile=CH5 SCRIPT A dfn=CH5
clines=100 elines=19 msglevel=1 prperiod=256 lnpr=
   Shows how the input command was parsed.
CMPEXP (C) Copyright IBM Corporation 1993
File CH5 SCRIPT A will be compressed using CH5 CEDICT41 A
Reading in dictionary
Compressing; compression=((concatenated_output_bits/8)/input_characters)*100%.
Value in parentheses is (output_bytes_per_line/input_characters)*100%.
First of last three nmbrs is chars in cur. 256 lines, next is all chars so far,
```

```
next is dict entries used so far (initialized with alphabet entries and SDs)
Line 256 37.72% (38.25%) for 256, 37.72% (38.25%) for all; 12757 12757 1874
Line 512 34.71% (35.60%) for 256, 36.46% (37.15%) for all; 9127 21884 2260
   The 36.46% is calculated from the concatenation of the index
   symbols produced during the compression of the first 512 lines.  The
   37.15% is calculated by rounding up to a whole number of bytes the
   concatenation of index symbols produced during the compression of each line.
Line 768 30.43% (31.00%) for 256, 34.29% (34.93%) for all; 12322 34206 2492
Line 1024 34.39% (34.85%) for 256, 34.32% (34.91%) for all; 15913 50119 2851
Line 1280 30.04% (30.66%) for 256, 33.50% (34.09%) for all; 11894 62013 3011
Line 1536 30.04% (30.57%) for 256, 32.90% (33.48%) for all; 13037 75050 3117
Line 1792 30.86% (31.32%) for 256, 32.61% (33.18%) for all; 12229 87279 3264
Line 2048 36.42% (36.88%) for 256, 33.11% (33.66%) for all; 13139 100418 3385
Line 2304 28.83% (29.46%) for 256, 32.70% (33.26%) for all; 10592 111010 3516
Line 2560 27.37% (28.03%) for 256, 32.20% (32.77%) for all; 11625 122635 3569
Line 2816 28.20% (28.70%) for 256, 31.83% (32.39%) for all; 12405 135040 3638
Line 3072 27.57% (28.11%) for 256, 31.44% (32.00%) for all; 13500 148540 3710
Line 3328 31.59% (32.07%) for 256, 31.46% (32.01%) for all; 13471 162011 3768
Line 3584 26.68% (27.27%) for 256, 31.11% (31.66%) for all; 12668 174679 3787
Line 3840 27.79% (28.42%) for 256, 30.91% (31.47%) for all; 10888 185567 3808
Line 4096 31.05% (31.72%) for 256, 30.92% (31.49%) for all; 11086 196653 3820
Line 4352 28.77% (29.23%) for 256, 30.77% (31.33%) for all; 14412 211065 3841
Line 4608 28.64% (29.11%) for 256, 30.63% (31.19%) for all; 14934 225999 3846
Line 4864 30.96% (31.61%) for 256, 30.65% (31.21%) for all; 10386 236385 3858
Line 5120 29.95% (30.53%) for 256, 30.61% (31.17%) for all; 13038 249423 3873
Line 5376 27.93% (28.39%) for 256, 30.46% (31.02%) for all; 14449 263872 3885
Line 5632 28.21% (28.84%) for 256, 30.37% (30.93%) for all; 11255 275127 3898
Line 5888 27.62% (28.17%) for 256, 30.25% (30.81%) for all; 12752 287879 3924
Line 6144 28.81% (29.33%) for 256, 30.19% (30.75%) for all; 11752 299631 3929
Line 6400 27.10% (27.73%) for 256, 30.06% (30.62%) for all; 13303 312934 3936
Line 6656 26.96% (27.48%) for 256, 29.93% (30.48%) for all; 14267 327201 3938
Line 6912 29.33% (29.89%) for 256, 29.90% (30.46%) for all; 13212 340413 3951
Line 7168 30.30% (30.84%) for 256, 29.92% (30.47%) for all; 13100 353513 3961
Line 7424 29.78% (30.33%) for 256, 29.91% (30.47%) for all; 12488 366001 3969
Line 7680 29.36% (29.86%) for 256, 29.89% (30.45%) for all; 12808 378809 3978
Line 7936 31.25% (31.79%) for 256, 29.94% (30.49%) for all; 11739 390548 3980
Line 8192 31.81% (32.15%) for 256, 30.01% (30.56%) for all; 16398 406946 4000
Line 8448 33.90% (34.45%) for 256, 30.12% (30.67%) for all; 12019 418965 4012
Line 8704 35.04% (36.12%) for 256, 30.22% (30.77%) for all; 8411 427376 4020
Line 8801 39.22% (39.94%) for 97, 30.32% (30.88%) for all; 4765 432141 4020
   The following lines are written only when msglevel is 2, but they
   are shown here anyway.
Index symbols of lengths 1-20:
12458 12773 13293 9007 9022 6828 6721 6029 3353 2429 1728 1263 736 557 138 232
95 136 76 60
   For example, there are 9007 index symbols that represent
   a character symbol of length four characters.
Index symbols of lengths 21-40:
45 40 33 52 18 34 8 15 8 8 9 17 15 8 38 24 0 2 2 10
Index symbols of lengths 41-60:
0 8 0 7 0 41 0 7 0 0 0 1 0 7 0 11 0 0 0 4
Index symbols of lengths 61-80:
0 2 0 4 0 4 0 1 4 1 0 0 0 0 24 0 0 0 0 0
   There are 24 index symbols that represent character symbols of
   length 75 characters despite that the dictionary is only 28 levels
   deep.  This is due to the additional extension characters in the
   entries.  The following lines are written when msglevel is 1.
Input chars=432141; conc. output bits=1048140; bytes in output lines=133427
Bits per index symbol=12
Expanding
Done
```

Appendix M. Converting fork() + exec() to spawn()

This appendix contains two sections. The first section provides two conversion examples. “Factors to Consider When Converting” on page 618 provides a detailed description of using the `spawn()` function and identifies factors to consider when doing a conversion from `fork()`.

Conversion Examples

This section provides two examples of programs that use the `fork()` and `exec()` functions and shows how the same operations can be achieved by converting the programs to use `spawn()`.

Example 1

In this code fragment, an application uses the `fork()` function to create a child process. The child then becomes the process group leader of a new process group that runs in the foreground and invokes the `exec()` function to run another application.

fork() version

```
/*************************************************************/
/*                                                           */
/*       Issue fork to create a child that gets into its     */
/*       own process group, puts itself in the foreground,   */
/*       then issues exec().                                 */
/*                                                           */
/*************************************************************/
 if ((pid = fork()) == -1) {
   return(-1);
 }
 else {
   if (pid == 0) {                              /* child */
     if (setpgid((pid_t) 0, (pid_t) 0) == -1) {
       exit(-1);
     }
     else {
       if (tcsetpgrp(cterm_fd, getpgrp()) == -1) {
         exit(-1);
       }
       else {
         execve("/prog", prog_args, my_env);
         exit(-1);
       }
     }
   }
   else {                                       /* parent */
     retpid = waitpid(pid, &status, 0);
   }
 }
```

spawn() version

```
/*************************************************************/
/*                                                           */
/*       Issue spawn to create a child process that is       */
/*       in its own process group and is in the foreground.  */
/*                                                           */
/*************************************************************/

  inherit.flags = SPAWN_SETGROUP | SPAWN_SETTCPGRP;
  inherit.pgroup = SPAWN_NEWPGROUP;
  inherit.ctlttyfd = cterm_fd;

  if ((pid =
    spawn("/prog", 0, NULL, &inherit, prog_args, my_env)) == -1) {
    return(-1);
  }
  else {
```

```
      retpid = waitpid(pid, &status, 0);
    }
  }
```

Example 2

This example demonstrates how to convert an application from using `fork()` to using `spawn()` when the parent's signal environment differs from the child's. In this example, the parent must ignore specific signals. The child should ignore these signals only if they were being ignored at the time this code fragment was invoked, otherwise they should be set to the default action. In addition, the parent must block the SIGCHLD signal, while the child must run with the signal mask that was in place at the time this code received control.

fork() version

```
/*********************************************************************/
/*    Description:                                                   */
/*      Execute the command specified by the string pointed to       */
/*      by 'cmd'.  The environment of the executed command shall     */
/*      be as if a child process were created using the fork()       */
/*      function, and the child process invoked the 'sh'             */
/*      utility using the execl() function as follows:               */
/*                                                                   */
/*          execl(<shell path>, "sh", "-c", cmd, (char *)0);         */
/*                                                                   */
/*      This function shall ignore the SIGINT and                    */
/*      SIGQUIT signals, and block the SIGCHLD signal, while         */
/*      waiting for the command to terminate.                        */
/*      This function shall not return until the child               */
/*      process has terminated.                                      */
/*********************************************************************/
         int              stat;
         pid_t            pid;
         sigset_t         saveblock;
         struct sigaction sa;
         struct sigaction savintr;
         struct sigaction savequit;

         sa.sa_handler = SIG_IGN;
         sigemptyset(&sa.sa_mask);
         sa.sa_flags = 0;
         sigemptyset(&savintr.sa_mask);
         sigemptyset(&savequit.sa_mask);
         sigaction(SIGINT, &sa, &savintr);
         sigaction(SIGQUIT, &sa, &savequit);
         sigaddset(&sa.sa_mask, SIGCHLD);
         sigprocmask(SIG_BLOCK, &sa.sa_mask, &saveblock);

         if ((pid = fork()) == 0) {               /* child */
            sigaction(SIGINT, &savintr, 0);
            sigaction(SIGQUIT, &savequit, 0);
            sigprocmask(SIG_SETMASK,&saveblock, 0);
            execl("/bin/sh", "sh", "-c", cmd, (char *)0);
            _exit(127);
         }
         if (pid == -1) {
            stat = -1;
         }
         else {
            while (waitpid(pid, &stat, 0) == -1) {
               if (errno != EINTR) {
                  stat = -1;
                  break;
               }
            }
         }
         sigaction(SIGINT, &savintr, 0);
         sigaction(SIGQUIT, &savequit, 0);
         sigprocmask(SIG_SETMASK, &saveblock, 0);
         return(stat);
```

spawn() version

```
/*********************************************************************/
/*    Description:                                                   */
/*      Execute the command specified by the string pointed to       */
/*      by 'cmd'.  The environment of the executed command shall     */
/*      be as if a child process were created using the fork()       */
/*      function, and the child process invoked the 'sh'             */
/*      utility using the execl() function as follows:               */
/*                                                                   */
/*          execl(<shell path>, "sh", "-c", cmd, (char *)0);         */
/*                                                                   */
/*      This function shall ignore the SIGINT and                    */
/*      SIGQUIT signals, and block the SIGCHLD signal, while         */
/*      waiting for the command to terminate.                        */
/*      This function shall not return until the child               */
/*      process has terminated.                                      */
/*                                                                   */
/*      This version uses spawn() instead of fork(), exec().         */
/*                                                                   */
/*      Set up the spawn() inheritance structure so that:            */
/*      1) SIGQUIT and SIGINT will be set to their default           */
/*         actions in the child, if the process had not set them     */
/*         to SIG_IGN prior to calling this function. If either had  */
/*         been set to SIG_IGN prior to the call, do nothing         */
/*         because signals set to be ignored in the parent will      */
/*         also be ignored in the child.                             */
/*      2) The child will inherit the signal mask of the parent,     */
/*         before the parent invoked this function.                  */
/*                                                                   */
/*      Get the pointer to environ and pass it to spawn(), so that   */
/*      the child inherits the parent's environment variables.       */
/*                                                                   */
/*********************************************************************/
         int               stat;
         pid_t             pid;
         struct sigaction  sa;
         struct sigaction  savintr;
         struct sigaction  savequit;

         struct inheritance inherit;
         char *args[4] = { "sh", "-c", NULL, NULL };
         extern char **environ;

         args[2] = (char *)cmd;
         inherit.flags = 0;
         sigemptyset(&inherit.sigdefault);

         sa.sa_handler = SIG_IGN;
         sigemptyset(&sa.sa_mask);
         sa.sa_flags = 0;
         sigemptyset(&savintr.sa_mask);
         sigemptyset(&savequit.sa_mask);
         sigaction(SIGINT, &sa, &savintr);
         sigaction(SIGQUIT, &sa, &savequit);
/*********************************************************************/
/*|  We need to add SIGCHLD to our process signal mask, but we only  */
/*|  want SIGCHLD to be in the signal mask of the child if it is     */
/*|  currently in the signal mask. So we need to obtain the current  */
/*|  signal mask to be passed in the sigmask field of the            */
/*|  inheritance structure.                                          */
/*********************************************************************/
         sigaddset(&sa.sa_mask, SIGCHLD);
         sigprocmask(SIG_BLOCK, &sa.sa_mask, &inherit.sigmask);
         inherit.flags |= SPAWN_SETSIGMASK;
/*********************************************************************/
/*|  If SIGINT and SIGQUIT were not being ignored at the time we     */
/*|  were invoked, we need to pass them in the sigdefault field      */
/*|  of the inheritance structure so that they will be set to the    */
/*|  default action in the child, and not be ignored, like in        */
/*|  the parent.                                                     */
/*********************************************************************/
         if (savintr.sa_handler != SIG_IGN)  {
            sigaddset(&inherit.sigdefault, SIGINT);
            inherit.flags |= SPAWN_SETSIGDEF;
         }
         if (savequit.sa_handler != SIG_IGN)  {
            sigaddset(&inherit.sigdefault, SIGQUIT);
            inherit.flags |= SPAWN_SETSIGDEF;
         }
```

```
if ((pid = spawn("/bin/sh", 0, NULL, &inherit,
                 (const char **)args,
                 (const char **)environ )) == -1) {
   switch (errno) {
      case ENOENT : stat = (127 << 8);
                    break;
      default :     stat = -1;
                    break;
   }
}
else {
   while (waitpid(pid, &stat, 0) == -1) {
      if (errno != EINTR) {
         stat = -1;
         break;
      }
   }
}
sigaction(SIGINT, &savintr, 0);
sigaction(SIGQUIT, &savequit, 0);
sigprocmask(SIG_SETMASK, &inherit.sigmask, 0);
return(stat);
```

Factors to Consider When Converting

The spawn() function creates a new process, loads a specified program into the new process, and begins execution. This creates a process that is nearly an exact duplicate of the process created when fork() and exec() are used together.

In the fork() plus exec() scenario, fork() creates a new child process that is nearly an exact duplicate of the original parent process. Both the parent and the child process continue execution at the return from fork(). The child process calls exec() to replace the program inherited from the parent and begin execution of the new process image file (program).

A point that must be understood in this scenario is that there may be additional processing performed by the child process *before* the exec() is done, or there may be no exec() at all. After the fork(), an exact copy of the parent process exists in the child process. All memory maps, variables, pointers, and so on, exist in the child just as they did in the parent. If you examine the buffers in the child, you can see exactly what was in the parent before the fork(). This allows the child to refer to these as if nothing has happened. The child can do one of the following:

- Continue processing the program inherited from the parent
- Do some additional processing, possibly in preparation for the exec()
- Immediately do an exec()

In contrast, when spawn() is executed there is no intermediate stage in the process. The program started by the exec() initiated by the child in the scenario described above is immediately available; however, all memory maps, variables, and so on, are lost to the child process. If any information in the parent process is required by the child, that information must be provided by an alternative method.

Inheritance

A program or "process image file" loaded using spawn() inherits certain characteristics of the parent process. Because spawn() is effectively a functional combination of fork() and exec(), the inherited attributes are an accumulation of the attributes of the two functions. Table 81 on page 618 illustrates this relationship. By following across the table from left to right, you will find in nearly every case that each characteristic of spawn() is shown by either fork() or exec() or both.

Table 81. Comparison of spawn() Attributes to fork() and exec()

Attribute	fork()	exec()	spawn()
Process PID	Changed	Not Changed	Changed
Process Group ID	Not Changed	Not Changed	Not Changed [1]

Table 81. Comparison of spawn() Attributes to fork() and exec() (continued)

Attribute	fork()	exec()	spawn()
Parent PID	Changed	Not Changed	Changed
Session Membership	Not Changed	Not Changed	Not Changed
Real UID	Not Changed	Not Changed	Not Changed
Real GID	Not Changed	Not Changed	Not Changed
Effective UID	Not Changed	May Change [2]	May Change [2]
Effective GID	Not Changed	May Change [2]	May Change [2]
Saved-Set UID	Not Changed	May Change [3]	May Change [3]
Saved-Set GID	Not Changed	May Change [3]	May Change [3]
Supplemental GIDs	Not Changed	Not Changed	Not Changed
Time Left on Alarms	Reset to 0	Not Changed	Reset to 0
Process Signal Mask	Not Changed	Not Changed	Not Changed [4]
Pending Signals	Empty Set	Not Changed	Empty Set
Signals Set to Be Caught	Not Changed	Caught set to SIG_DFL	Caught set to SIG_DFL
Signals Set to Default Action	Not Changed	Not Changed	Not Changed
Signals Set to Ignore	Not Changed	Not Changed	Not Changed [5]
Process Times	Cleared to 0	Not Changed	Cleared to 0
Open File Descriptors	Closed if FD_CLOFORK set [6]	Closed if FD_CLOEXEC set [6]	Closed if FD_CLOEXEC, FD_CLOFORK, or SPAWN_FDCLOSED [6]
Open Directory Streams	Duplicated	Closed	Closed
File Locks	Not Inherited	Not Changed if file still open	Not Inherited
Current Working Directory	Not Changed	Not Changed	Not Changed
Root Directory	Not Changed	Not Changed	Not Changed
File Creation Mask	Not Changed	Not Changed	Not Changed
Memory Locks	Not Changed	Released	Released

Table 81. Comparison of spawn() Attributes to fork() and exec() (continued)

Attribute	fork()	exec()	spawn()

Note:

1. The process group ID may be changed with the *inherit* parameter. See Inheritance Structure.
2. When a new process is created by `spawn()`, if the set-user-ID mode bit is set in the new process image, the effective user ID is set to the owner of the new process image file. If the set-group-ID mode bit is set, the effective group ID of the new process image file is set to the group of the new process image file.
3. The effective user ID and effective group ID of the new process are saved as the saved-set UID and saved-set GID.
4. The signal mask can be changed using the *inherit* parameter. See Inheritance Structure.
5. The signals forced to the default action in the child process can be controlled with the *inherit* parameter. See Inheritance Structure.
6. FD_CLOEXEC and FD_CLOFORK can be set by `fcntl()` on an open file. If this flag is set, the flagged file is closed in the new process after the `spawn()`. SPAWN_FDCLOSED is a value set in the *map[]* array of `spawn()`. If an array element has this value, that file descriptor is closed in the new process. Every element of the *map[]* array must have a valid value. If the array element does not have a mapped array element, it must contain the constant SPAWN_FDCLOSED.

Parameters

The `spawn()` function provides several parameters (described below) that allow various types of information to be passed to the child process.

```
include <spawn.h>

pid_t spawn( const char *path,
             const char fd_count,
             const char fd_map[],
             const struct inheritance *inherit,
             const char *argv[],
             const char *envp[]);

pid_t spawnp( const char *file,
              const char fd_count,
              const char fd_map[],
              const struct inheritance *inherit,
              const char *argv[],
              const char *envp[]);
```

The `spawn()` function comes in two forms, `spawn()` and `spawnp()`. The function calls are identical, with the exception of the method used to locate the process image file (program). The *path* parameter of `spawn()` identifies the new process image file to be executed. The *file* parameter of `spawnp()` is used to construct a path name that identifies the new process image file. The path is determined as follows. If the file name contains a '/', the file name is the path name of the new process image file. If not, the directories passed in the PATH environment variable are used as a prefix to the file name until either the file is found or all directories are searched.

Note: The PATH environment variable can contain several directory paths.

The remaining parameters for both functions are identical and allow considerable flexibility when calling the new process image file.

The parameters provide four distinct services:

- Remapping file descriptors
- Altering certain attributes of the new process
- Passing a parameter list to the process image file being loaded
- Passing environment variables to the new process image.

These services are described in more detail below.

Remapping of File Descriptors – fd_count, fd_map[]

Remapping of file descriptors provides programmers with a very powerful tool. It allows the program using `spawn()` to create the child process to exercise complete control over:

- Which open files (file descriptors) are inherited by the child
- The file descriptor number each inherited file will have.

Remapping is controlled by two parameters, *fd_count* and *fd_map[]*.

The *fd_count* parameter specifies the file descriptors that are mapped to the child process. All file descriptors above the value in *fd_count* are closed in the child process.

Note:

1. In the C environment, file descriptors 0-2 are typically used for stdin, stdout, and stderror.
2. If the pointer to *fd_map* is NULL, no mapping takes place and all file descriptors are inherited, except those with the FD_CLOEXEC or FD_CLOFORK attributes.

In conjunction with *fd_count*, the *fd_map[]* array maps parent file descriptors to child file descriptors.

The relationship between *fd_count* and *fd_map[]* is one to one; that is, there *must* be a map entry in *fd_map[]* for each file descriptor between zero (0) and *fd_count-1*. When `spawn()` is executed, file descriptors in the child are mapped using the information provided in *fd_map[]*. No file descriptors beyond the *fd_count* limit are inherited by the child process.

The following example illustrates the remapping of open files from Program A to Program B. The column on the left represents the file descriptors of interest in the parent process, the column on the right represents the remapped file descriptors in the child, and the center column represents the *fd_map[]* array elements 0-6.

```
    PARENT                                    CHILD
Prog A environment      fd_map[]           Prog B environment
                        fd_count = 7       fd in   fd from
                      array
   open fd           element#  content     child   parent
=============         ===============     ==================
     0                   0       0           0       0
     1                   1       1           1       1
     2                   2       2           2       2
     3                   3       3           3       3
     5                   4    S_FC           4     N/A
     .                   5      17           5      17
     .                   6      22           6      22
    17
     .
     .
    22

   S_FC = the SPAWN_FDCLOSED constant
   N/A  = the file descriptor is not open
```

The objective is to have open file descriptors 0-3, 17, and 22 in the parent mapped to file descriptors 0-3, 5, and 6 respectively in the child. All other file descriptors are closed in the child process.

The *fd_count* value, 7, ensures that all file descriptors above 6 are closed in the child. The array represented in the center column guides the remapping. Each array element number represents the child file descriptor of that number; that is, array element number 5 = child file descriptor 5, and the contents of the array element represents the parent file descriptor (in this case, 17) to be mapped into the child. Notice that any element not specifically mapped (such as element 4 in this example) contains the constant SPAWN_FDCLOSED. This value must be in each array element not mapped to a specific parent file descriptor or the results of the mapping are unpredictable. Open file descriptors in the parent that are not mapped (such as descriptor 5 in this example) are not represented in the child. File descriptors in the parent that are not open (such as descriptor 4 in this example) are not included in the mapping process.

Inheritance Structure – Used to Alter Attributes in the Child Process

The inheritance information provided to `spawn()` addresses process group membership, signal actions, signal masks, and session control.

The default behaviors (inherit.flags = 0) of the child process are:

- The process group ID of the child process is the process group ID of the parent process.
- Any signal set to be caught in the parent is established in the child with the default action (SIG_DFL). However, if the action in the parent is to ignore (SIG_IGN), the signal is ignored in the child as well.
- Signal mask information is inherited from the parent. Any signal blocked in the parent process is also blocked in the child process.

If the default actions are not acceptable, they can be changed using the *inherit* parameter of `spawn()`. Changing the attributes involves two steps:

1. Indicate which attributes are being changed in the inherit.flags member of the structure.
2. Provide the required information in the appropriate member of the inherit structure for the attributes being changed.

The primary point to remember when using this `spawn()` parameter is that everything starts with `inherit.flags`. The inherit function uses the contents of this member to determine what else to process. Only those functions indicated in this member of the inheritance structure are handled. Table 82 on page 622 shows which flags are associated with which members of the inheritance structure. The column at the left identifies the member of the inheritance structure. The columns to the right of .flags show the possible contents of that member. For example, if .flags has only SPAWN_SETSIGDEF set, then by going down the table in the SPAWN_SETSIGDEF column, the only member related is .sigdefault.

The notes associated with some table entries provide additional information that may be helpful.

Table 82. Inheritance Structure Usage

struct inherit	**flags controlling inheritance**			
.flags	**SPAWN_SETGROUP**	**SPAWN_SETSIGDEF**	**SPAWN_SETSIGMASK**	**SPAWN_SETTCPRGP**
.pgroup	Child's PGID [1]			
	SPAWN_NEWGROUP [2]			
.sigdefault		Signals that will be default [3]		
.sigmask			New signal blocking mask [4]	
.ctlttyfd				Controlling terminal file descriptor [5]

Note:

1. The child process takes the process group ID specified in the .pgroup member of 'struct inheritance'.
2. A new process group is created, with the PID of the child process as it's process group ID. The child process becomes the process group leader of the new process group.
3. The signals specified in the .sigdefault structure member have the default action (SIG_DEF) in the child process regardless of their state in the parent process. The state of the signals in the parent is not affected.
4. The signal blocking mask specified in this structure member is inherited by the child process.
5. This member contains the file descriptor of the session's controlling terminal. SPAWN_SETTCPRGP (.ctlttyfd) also puts the child's process group in the foreground.

Inheritance Conversion Tips

In programs being converted from `fork()` to `spawn()`, if any of the functions listed in Table 83 on page 623 are done in preparation for using `exec()` to execute a process image file, you should probably put the corresponding information into the inheritance structure.

Table 83. Inheritance Conversion Tips		
fork() setup function	**spawn() setup information**	
	associated struct member	**associated flag**
setpgrp()	.pgroup	SPAWN_SETGROUP
sigaction()	.sigdefault	SPAWN_SETSIGDEF
sigprocmask()	.sigmask	SPAWN_SETSIGMASK
tcsetpgrp()	.ctlttyfd	SPAWN_SETTCPRGP

Passing the Argument List to the Called Program – argv[]

The argument list is a character array. The first entry should point to the file name associated with the spawn() and the last entry must be the NULL pointer. The arguments are determined by the needs of the program being created by spawn().

This array is used to pass information required by the new process. This could include information that was available to the child process in the fork() scenario but is lost on a call to spawn(), such as variables, buffer contents, or information needed by the child unrelated to spawn(). Other information passed in the parameter list is purely dependent on the new process image file.

Passing Environment Variables – envp[]

The environment parameter is a list of character strings in the form *name=value*, where *name* is the name of the environment parameter and *value* is the value to which you want it set. These strings make up the environment of the new process image. The last entry in the environment list must be the NULL pointer.

If you want the child to inherit the environment of the parent, you can specify the external variable *environ* in the spawn() call.

Notices

This information was developed for products and services offered in the US. This material might be available from IBM in other languages. However, you may be required to own a copy of the product or product version in that language in order to access it.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you provide in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

The performance data and client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information may contain examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

COPYRIGHT LICENSE:

This information may contain sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Programming Interface Information

This document contains intended Programming Interfaces that allow the customer to write programs to obtain services of z/VM.

Trademarks

IBM, the IBM logo, and ibm.com® are trademarks or registered trademarks of International Business Machines Corp., in the United States and/or other countries. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on IBM Copyright and trademark information (https://www.ibm.com/legal/copytrade).

The registered trademark Linux is used pursuant to a sublicense from the Linux Foundation, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States and other countries.

Java™ and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Terms and Conditions for Product Documentation

Permissions for the use of these publications are granted subject to the following terms and conditions.

Applicability

These terms and conditions are in addition to any terms of use for the IBM website.

Personal Use

You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of IBM.

Commercial Use

You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Rights

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

IBM Online Privacy Statement

IBM Software products, including software as a service solutions, ("Software Offerings") may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user, or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering's use of cookies is set forth below.

This Software Offering does not use cookies or other technologies to collect personally identifiable information.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see:

- The section entitled **IBM Websites** at IBM Privacy Statement (https://www.ibm.com/privacy)
- 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/Support Facility on the Hardware Management Console (https://www.ibm.com/docs/en/SSLTBW_2.3.0/pdf/SC14-7580-02.pdf), SC14-7580
- Open Systems Adapter-Express ICC 3215 Support (https://www.ibm.com/docs/en/zos/2.3.0?topic=osa-icc-3215-support), SA23-2247
- Open Systems Adapter Integrated Console Controller User's Guide (https://www.ibm.com/docs/en/SSLTBW_2.3.0/pdf/SC27-9003-02.pdf), SC27-9003
- Open Systems Adapter-Express Customer's Guide and Reference (https://www.ibm.com/docs/en/SSLTBW_2.3.0/pdf/ioa2z1f0.pdf), SA22-7935

Performance Toolkit for z/VM

- *z/VM: Performance Toolkit Guide*, SC24-6302
- *z/VM: Performance Toolkit Reference*, SC24-6303

The following publications contain sections that provide information about z/VM Performance Data Pump, which is licensed with Performance Toolkit for z/VM.

- *z/VM: Performance*, SC24-6301. See z/VM Performance Data Pump.
- *z/VM: Other Components Messages and Codes*, GC24-6300. See Data Pump Messages.

RACF® Security Server for z/VM

- *z/VM: RACF Security Server Auditor's Guide*, SC24-6305
- *z/VM: RACF Security Server Command Language Reference*, SC24-6306
- *z/VM: RACF Security Server Diagnosis Guide*, GC24-6307
- *z/VM: RACF Security Server General User's Guide*, SC24-6308
- *z/VM: RACF Security Server Macros and Interfaces*, SC24-6309
- *z/VM: RACF Security Server Messages and Codes*, GC24-6310
- *z/VM: RACF Security Server Security Administrator's Guide*, SC24-6311
- *z/VM: RACF Security Server System Programmer's Guide*, SC24-6312
- *z/VM: Security Server RACROUTE Macro Reference*, SC24-6324

Remote Spooling Communications Subsystem Networking for z/VM

- *z/VM: RSCS Networking Diagnosis*, GC24-6316
- *z/VM: RSCS Networking Exit Customization*, SC24-6317
- *z/VM: RSCS Networking Messages and Codes*, GC24-6318
- *z/VM: RSCS Networking Operation and Use*, SC24-6319
- *z/VM: RSCS Networking Planning and Configuration*, SC24-6320

TCP/IP for z/VM

- *z/VM: TCP/IP Diagnosis Guide*, GC24-6328
- *z/VM: TCP/IP LDAP Administration Guide*, SC24-6329
- *z/VM: TCP/IP Messages and Codes*, GC24-6330
- *z/VM: TCP/IP Planning and Customization*, SC24-6331
- *z/VM: TCP/IP Programmer's Reference*, SC24-6332
- *z/VM: TCP/IP User's Guide*, SC24-6333

Prerequisite Products

Device Support Facilities

- Device Support Facilities (ICKDSF): User's Guide and Reference (https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/ickug00_v2r5.pdf), GC35-0033

Environmental Record Editing and Printing Program

- Environmental Record Editing and Printing Program (EREP): Reference (https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/ifc2000_v2r5.pdf), GC35-0152
- Environmental Record Editing and Printing Program (EREP): User's Guide (https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/ifc1000_v2r5.pdf), GC35-0151

Related Products

XL C++ for z/VM

- *XL C/C++ for z/VM: Runtime Library Reference*, SC09-7624
- *XL C/C++ for z/VM: User's Guide*, SC09-7625

z/OS

IBM Documentation - z/OS (https://www.ibm.com/docs/en/zos)

Index

Special Characters

Numerics

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Product Number: 5741-A09

Printed in USA

SC24-6256-73