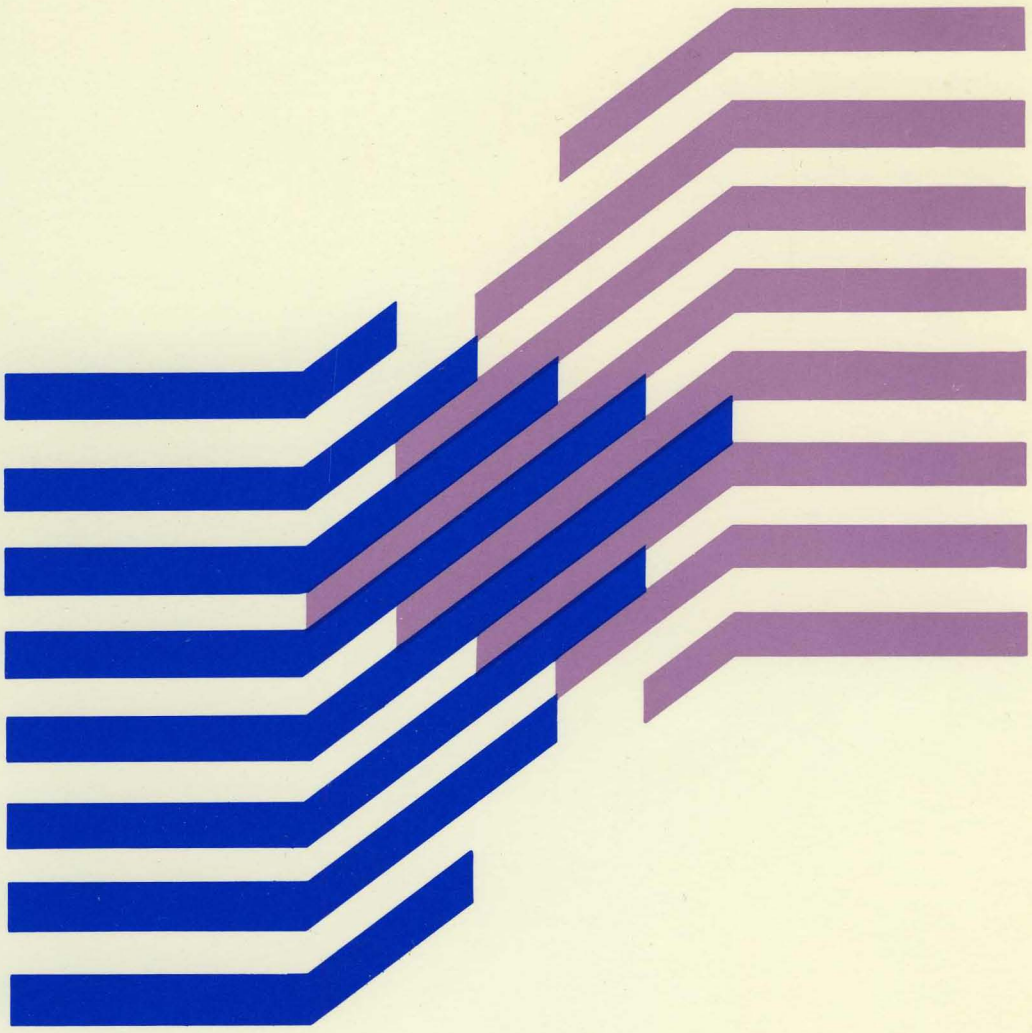
MVS / ESA

GC28-1644-1

Application Development Guide: Assembler Language Programs

MVS / ESA System Product:
JES2 Version 4
JES3 Version 4



MVS / ESA

GC28-1644-1

Application Development Guide: Assembler Language Programs

MVS / ESA System Product:
JES2 Version 4
JES3 Version 4

Note!

Before using this information and the product it supports, be sure to read the general information under "Notices" on page xii.

Production of This Book

This book was prepared and formatted using the IBM BookMaster document markup language.

Second Edition (March, 1991)

This is a major revision of, and obsoletes, GC28-1644-0 and Technical Newsletter GN28-1149. See the Summary of Changes for the changes made to this manual. Technical changes or additions to the text and illustrations are indicated by a vertical line to the left of the change.

This edition applies to Version 4 of MVS/ESA System Product 5695-047 or 5695-048 and to all subsequent releases and modifications until otherwise indicated in new editions or Technical Newsletters. Make sure you are using the correct edition for the level of the product.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

A form for reader's comments is provided at the back of this publication. If the form has been removed, address your comments to:

IBM Corporation, Department D58
PO Box 950
Poughkeepsie, N.Y. 12602
United States of America

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Figures

Notices

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates.

Any reference to an IBM licensed program or other IBM product in this publication is not intended to state or imply that only IBM's program or other product may be used. Any functionally equivalent program which does not infringe any of IBM's intellectual property rights may be used instead of the IBM product. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Commercial Relations, IBM Corporation, Purchase, NY 10577.

Programming Interfaces

This book is intended to help customers to code macros that are available to all assembler language programs. It contains guidance information needed to use the macros. This book documents general-use programming interfaces and associated guidance information provided by MVS/ESA System Product Version 4.

General-use programming interfaces allow the customer to write programs that request or receive the services of MVS/ESA System Product Version 4.

Trademarks

The following terms, **DENOTED BY AN ASTERISK (*)**, used in this publication, are trademarks of the IBM Corporation in the United States and/or other countries:

- BookMaster
- ESA/370
- Hiperspace
- IBM
- IPM
- MVS
- MVS/DFP
- MVS/ESA
- MVS/SP
- MVS/XA
- PSL
- System/370
- System/390
- VM
- VSE

About This Book

This book describes the operating system services that an unauthorized program can use. An unauthorized program is one that does not run in supervisor state, or have PSW key 0-7, or reside on an APF-authorized library. To use a service, the program issues a macro. A companion book, *MVS/ESA Application Development Reference: Services for Assembler Language Programs*, provides the detailed information for coding the macros.

Some of the topics discussed in this book are also discussed in *MVS/ESA Application Development Guide: Authorized Assembler Language Programs* and *MVS/ESA Application Development Reference: Services for Authorized Assembler Language Programs*. However, the services and macros in those books are for authorized programs.

Who Should Use This Book

This book is for the programmer who is coding in assembler language, and who needs to become familiar with the operating system and the services that programs running under it can invoke.

The book assumes that the reader understands system concepts and writes programs in assembler language. Assembler language programming is discussed in the following books:

- *Assembler H Version 2 Application Programming Guide*, SC26-4036
- *Assembler H Version 2 Application Programming: Language Reference*, GC26-4037

Callable Services for High-Level Languages, GC28-1639 contains information about operating system services for high-level language programmers.

How To Use This Book

This book is one of the set of application development books for MVS. This set describes how to develop applications in assembler language or high-level languages, such as C, FORTRAN, and COBOL. The following figures show how this book fits in with the others in the set:

Figure 1-1. Application Development Books for Assembler Language Programs

Book	Use this book to:
Assembler Programming Guide, GC28-1644	Find out how to use system services provided by macros available to all assembler language programs. If you are relatively new to assembler language programming, this book is a good place to start.
Assembler Programming Reference, GC28-1642	Learn how to code macros that are available to all assembler language programs. This book is for all assembler language programmers.
Authorized Assembler Programming Guide, GC28-1645	Find out how to use system services provided by macros that are available to programs running in supervisor state or with PSW key 0-7 or that are APF-authorized programs. This book is for experienced assembler language programmers; it assumes, for example, that you are familiar with the information in *Assembler Programming Guide.*
Authorized Assembler Programming Reference, GC28-1647 to 1650	Learn how to code macros that are available to programs running in supervisor state or with PSW key 0-7 or that are APF-authorized programs. This book is for experienced assembler language programmers.
Extended Addressability Guide, GC28-1652	Find out how to extend the storage available to programs through the use of access registers, cross memory services, data spaces, and hiperspaces. This book is for experienced assembler language programmers.
Hiperbatch* Guide, GC28-1673	Find out whether your installation's batch applications can benefit from Hiperbatch and how to use Hiperbatch. Using this book does not require a knowledge of assembler language programming.
Batch LSR Guide, GC28-1672	Find out whether your installation's batch applications can benefit from batch LSR subsystem and how to use batch LSR subsystem. Using this book does not require a knowledge of assembler language programming.

* Hiperbatch is a trademark of the IBM Corporation.

Figure 1-2. Application Development Books for High-Level Language Programs

Book	Use this book to:
Callable Services for High-Level Languages, GC28-1639	Find out how to use and code program CALLs for specific MVS services. This book is for programmers who code programs in high-level languages.

Figure 1-3. Application Development Books for Both Assembler and High-Level Language Programs

Book	Use this book to:
Authorized Callable Services, GC28-1112	Find out how to use and code program CALLs for APPC/MVS system services that are intended for transaction schedulers written in assembler or high-level languages. This book is for programmers who write transaction schedulers other than the one APPC/MVS provides.
Writing Transaction Programs for APPC/MVS, GC28-1121	Find out how to use and code program CALLs for APPC/MVS communication services. This book is for programmers who code APPC/MVS transaction programs in assembler or high-level languages.

Where to Find More Information

Where necessary, this book references information in other books, using shortened versions of the book title. The following table shows the full titles and the order numbers:

Short Title Used in This Book	Title	Order Number
Assembler H Version 2 Application Programming: Language Reference	*Assembler H Version 2 Application Programming: Language Reference*	GC26-4037
Assembler Programming Guide	*MVS/ESA Application Development Guide: Assembler Language Programs*	GC28-1644 THIS BOOK
Assembler Programming Reference	*MVS/ESA Application Development Reference: Services for Assembler Language Programs*	GC28-1642
Authorized Assembler Programming Guide	*MVS/ESA Application Development Guide: Authorized Assembler Language Programs*	GC28-1645
Authorized Assembler Programming Reference	*MVS/ESA Application Development Reference: Services for Authorized Assembler Language Programs, Volumes 1 - 4*	GC28-1647 GC28-1648 GC28-1649 GC28-1650
DFP GIM	*MVS/DFP Version 3 Release 3 General Information*	GC26-4552
Diagnosis: Data Areas	*MVS/ESA Diagnosis: Data Areas*	LY28-1821 thru LY28-1825
Diagnosis: Using Dumps and Traces	*MVS/ESA Diagnosis: Using Dumps and Traces*	LY28-1813
Extended Addressability Guide	*MVS/ESA Application Development Guide: Extended Addressability for Authorized Programs*	GC28-1652
Hiperbatch Guide	*MVS/ESA Application Development Guide: Hiperbatch*	GC28-1673

Short Title Used in This Book	Title	Order Number
Batch LSR Guide	*MVS/ESA Application Development Guide: Batch Local Shared Resources Subsystem*	GC28-1672
IBM System/370 Vector Operations	*IBM System/370 Vector Operations*	SA22-7125
IPCS Command Reference	*MVS/ESA Interactive Problem Control System (IPCS) Command Reference*	GC28-1632
IPCS User's Guide	*MVS/ESA Interactive Problem Control System (IPCS) User's Guide*	GC28-1631
JCL Reference	*MVS/ESA JCL Reference*	GC28-1654
JCL User's Guide	*MVS/ESA JCL User's Guide*	GC28-1653
JES3 Commands	*MVS/ESA Operations: JES3 Commands*	GC23-0074
MVS Conversion Notebook for Version 4	*MVS/ESA Conversion Notebook for MVS/ESA System Product Version 4*	GC28-1608
MVS Initialization and Tuning Guide	*MVS/ESA Initialization and Tuning Guide*	GC28-1634
MVS Initialization and Tuning Reference	*MVS/ESA Initialization and Tuning Reference*	GC28-1635
Installation Exits	*MVS/ESA Installation Exits*	GC28-1637
NLS Reference Manual	*NLS Reference Manual*	SE09-8002
Planning: Operations	*MVS/ESA Planning: Operations*	GC28-1625
Planning: Problem Determination and Recovery	*MVS/ESA Planning: Problem Determination and Recovery*	GC28-1629
Planning: Global Resource Serialization	*MVS/ESA Planning: Global Resource Serialization*	GC28-1621
Principles of Operation[1]	*ESA/370* Principles of Operation* *ESA/390* Principles of Operation*	SA22-7200 SA22-7201
Problem Determination Guide	*MVS/ESA Problem Determination Guide*	GC28-1667
Routing and Descriptor Codes	*MVS/ESA Routing and Descriptor Codes*	GC28-1666
Service Aids	*MVS/ESA Service Aids*	GC28-1669
System Commands	*MVS/ESA System Commands*	GC28-1626
System Messages	*MVS/ESA System Messages, Volumes 1 - 3*	GC28-1656 GC28-1657 GC28-1658
System Modifications	*MVS/ESA System Modifications*	GC28-1636
TSO/E Version 2 Programming Services	*TSO Extensions Version 2 Programming Services*	SC28-1875

1 Use the appropriate *Principles of Operation* book for the hardware you have installed.

* ESA/370 is a trademark of the IBM Corporation.

* ESA/390 is a trademark of the IBM Corporation.

Notes:

1. All references to Assembler H in this publication indicate the program product Assembler H Version 2 (5668-962).
2. All references to RMF in this publication indicate the program product Resource Measurement Facility (5685-029).

Summary of Changes

Summary of Changes
for GC28-1644-1
MVS/ESA* System Product Version 4 Release 2

This major revision consists of changes to support MVS/ESA System Product Version 4 Release 2.

New Information

- CSREVW allows your program to view an object and access it sequentially.
- REFPAT allows your program to identify a large data area and tell the system how the program will be referencing that area.
- The IOCINFO macro obtains the MVS I/O configuration token.
- The UCBSCAN macro scans UCBs and obtains UCB copies.
- The EDTINFO macro obtains information from the eligible device table (EDT).

Changed Information

- PFCOUNT parameter on the DIV macro allows you to request that the system preload up to 255 pages of virtual storage at a page fault.
- MVS now allows a problem state program with PSW key 8 - F to add a SCOPE=SINGLE data space to the PASN-AL, and to assign data space ownership to its job step task.

Summary of Changes
for GC28-1644-0
as updated by Technical Newsletter GN28-1149

Changed Information: This technical newsletter contains services updates.

Summary of Changes
for GC28-1644-0
MVS/ESA System Product Version 4 Release 1

This book contains information previously presented in *MVS/ESA Application Development Guide*, GC28-1821-2, which supports MVS/SP Version 3 Release 1.3.

This revision also includes minor maintenance and editorial changes.

Technical changes or additions to the text and illustrations are indicated by a vertical line to the left of the change.

The following summarizes the changes to that information.

* MVS/ESA is a trademark of the IBM Corporation.

New Information

- The CONVCON macro enables you to convert console IDs to names, and console names to IDs.
- A new resource name list (RNL) parameter on the DEQ and ENQ macros specifies whether RNL processing is to occur for a resource or resources.
- The LINKAGE=SYSTEM option on the TIME macro allows an application to use the DATETYPE parameter to specify the format for the returned date.
- The STCKCONV macro converts a TOD clock value to time of day and date in various formats.
- The STCKSYNC macro obtains the TOD clock contents and determines if the clock is synchronized with an external time reference (ETR).
- The SYMRBLD macro generates code to build a symptom record and logs it on the SYS1.LOGREC data set. IBM recommends using this macro instead of SYMREC.
- The SYMREC macro updates a symptom record with system environment information and logs the symptom record in the SYS1.LOGREC data set.
- The chapter on translating messages explains how to use the MVS message service to obtain translated versions of system or application messages.
- The WTO and WTOR macros have been enhanced to provide new methods for you to specify message text, and to provide increased flexibility for message queuing.
- A new chapter was added on 31-bit addressing. This information was previously presented in *MVS/ESA System Programming Library: Application Development 31-Bit Addressing*, GC28-1820-0.

Changed Information: The DIV macro section on restrictions for the SAVE and RESET services has been updated.

Moved Information: The reference information from the Callable Services section was moved to the *Assembler Programming Reference* book.

Deleted Information: None.

Terminology

External time reference (ETR) is the MVS generic name for the IBM Sysplex Timer (9037).

Chapter 1. Introduction

The system controls the flow of work through the computer so that all programs obtain a fair share of the processing. To make efficient use of the system, you must understand the services that the system provides and observe the programming conventions for their use.

Linkage Conventions — A program must follow register and save area conventions when it is called by another program or when it calls another program. These conventions ensure that the programs can successfully pass control to each other while preserving the register contents and the parameter data required for successful execution.

Subtask Creation and Control — Because the system can handle small programs easier than large ones, a large program might execute faster if you divide it into parts, called tasks. By following the appropriate conventions, you can break your programs into tasks that compete more efficiently for the resources of the system.

Program Management — Program residence and addressing modes are discussed in this chapter, as well as the linkage between programs. Save areas, addressability, and conventions for passing control from one program to another are also discussed.

Understanding 31-bit Addressing — 31-bit addressing terms are defined in this chapter. Read this chapter before writing new programs or modifying existing programs to use 31-bit addresses.

Resource Control — Anything necessary for program execution, such as a table, a storage device, or another program, can be considered a resource. If a program depends on an event that occurs in another program, it might need to defer part of its execution until the event, which is considered a resource, is completed. Because many programs might need the same resource, and because some resources can only be used by one program at a time, synchronization is often necessary. Resource control helps to regulate access to the resources that your programs depend on for successful execution.

Program Interruption and Termination — When your program comes to a successful end of execution, the ending is called a normal termination. When the system stops your program because you made an error, the ending is called an abnormal termination. Before your program terminates, it might be interrupted. You can write routines that get control when your program terminates normally or abnormally, or that get control when your program is interrupted, and you can specify the conditions under which these routines are to be executed.

Dumping Services — If your program makes serious errors, the system terminates it. If you request it, the system generates a dump to accompany the termination, and the resulting dump is called called an abend dump. You can also request another type of dump, called a snap dump. Programs can request a snap dump at any time, and they can specify the source, the format, and the destination of the information in the dump.

Virtual Storage Management — The system combines central storage and auxiliary storage to make the addressable memory appear larger than it really is. The apparent memory capacity is called virtual storage. By managing storage in this

way, the system relaxes the size limit on programs and data. The storage that the system gives to each related group of programs is called an address space. As a program executes, its storage requirements might vary. Conventions described in this chapter allow a program to obtain any extra storage it might require, and to return storage that is no longer required.

Callable Cell Pool Services — Callable cell pool services manage user-obtained areas of virtual storage efficiently, provide high performance service, and allow you to use storage in both address spaces and data spaces. This chapter describes callable cell pool services and helps you make the decision between using the CPOOL macro or callable cell pool services.

Data-In-Virtual — By using a simple technique that lets you create, read, or update external storage data without the traditional GET and PUT macros, you can write programs that use very large amounts of this type of data. The data, which is not broken up into individual records, appears in your virtual storage all at once. This technique also provides better performance than the traditional access methods for many applications.

Using Access Registers — If you need to access data in a data space, you need to use the set of registers called "access registers" and be in the address space control (ASC) mode called "AR mode". This chapter helps you access data in data spaces and use the system services while you are in AR mode.

Data Spaces and Hiperspaces — If you need more virtual storage than a single address space allows, and if you want to prevent other users from accessing this storage, you can use data spaces and hiperspaces.

Window Services — Window services enable assembler language programs to access or create permanent or temporary data objects. By invoking the service programs provided by window services, a program can:

- Read or update an existing data-in-virtual object
- Create and save a new permanent data-in-virtual object
- Create and use a temporary data-in-virtual object

Processor Storage Management — The system administers the use of processor storage (that is, central and expanded storage) and directs the movement of virtual pages between auxiliary storage and central storage in page-size blocks. You can release virtual storage contents, load virtual storage areas into central storage, and page out virtual storage areas from central storage. Reference pattern services allow programs to define a reference pattern for a specified area that the program is about to reference.

Timing and Communication — The system has an internal clock. Your program can use it to obtain the date and time, or use it as an interval timer. You can set a time interval, test how much time is left in an interval, or cancel it. Communication services let you send a message to the system operator, to a TSO terminal, or to the system log.

MVS Message Service — The MVS message service (MMS) enables you to display MVS or MVS-based application messages that have been translated from U.S. English into a foreign language. The service also allows application programs to store messages in and retrieve them from the MMS run-time message file.

Data Compression — Data compression and expansion services allow you to compress certain types of data so that the data occupies less space while you are not using it. You can then restore the data to its original state when you need it.

Accessing Unit Control Blocks — Each device in a configuration is represented by a unit control block (UCB). This chapter contains information about scanning UCBs and detecting I/O configuration changes.

IMPORTANT -------- READ THIS

As you read the book, keep in mind how the book uses the following terms:

- The term **registers** means general purpose registers. In sections where general purpose registers might be confused with other kinds of registers (such as access registers), the book uses the longer term **general purpose registers**.
- Unless otherwise specified, the **address space control (ASC) mode** of a program is primary mode.

Chapter 2. Linkage Conventions

Linkage conventions are the register and save area conventions a program must follow when it receives control from another program or when it calls another program. It is important that all programs follow the linkage conventions described here to ensure that the programs can successfully pass control from one to the other while preserving register contents and parameter data that they need to run successfully.

One program can invoke another program through any one of the following branch instructions or macros:

- BALR, BASR, or BASSM instructions
- LINK, LINKX, XCTL, XCTLX, and CALL macros

The program that issues the branch instruction or the macro is the **calling program**. The program that receives control is the **target program**. A program should follow these conventions when it:

- Receives control from a calling program
- Returns control to the calling program
- Calls another program

The PC instruction provides another means of program linkage. Linkage conventions for the PC instruction are described in *MVS/ESA Application Development Guide: Extended Addressability*.

In this chapter, programs are classified by their address space control (ASC) mode as follows:

- A **primary mode program** is one that executes all its instructions in primary ASC mode and does not change the contents of ARs 2 through 13.
- An **AR mode program** is one that executes one or more instructions in AR mode or it changes the contents of ARs 2 through 13. A program that switches from one mode to another is considered to be an AR mode program. A program that runs in AR mode can access data that is outside its primary address space.

The ASC mode at the time a program issues the call determines whether addresses passed by the program must be qualified by access list entry tokens (ALETs). An ALET identifies the address space or data space that contains the passed addresses. An ALET-qualified address is an address for which the calling program has provided an ALET. The ASC mode at the time of the call also determines whether the program can call a primary mode program or an AR mode program.

- A calling program that is in primary mode at the time of the call can call either another primary mode program or an AR mode program. Addresses passed by the calling program are not ALET-qualified.
- A calling program that is in AR mode at the time of the call can call only another AR mode program. Addresses passed by the calling program are ALET-qualified.

An AR mode program can call a primary mode program, but the calling program must first switch to primary mode and then follow the linkage conventions for a primary mode caller. Addresses passed by the calling program cannot be ALET-qualified.

When one program calls another, the target program receives control in the caller's ASC mode at the time the call was made. If the calling program is in AR mode at the time of the call, the target program receives control in AR mode. If the calling program is in primary mode at the time of the call, the target program receives control in primary mode. After a target program receives control, it can switch its ASC mode by issuing the Set Address Control (SAC) instruction. For more information on ASC mode, see Chapter 12, "Using Access Registers" on page 12-1.

Saving the Calling Program's Registers

At entry, all target programs save the caller's registers; at exit, they restore those registers. The two places where a program can save registers are in a **caller-provided save area** or in a **system-provided linkage stack**. The ASC mode of the target program determines how the target program saves the registers. A primary mode program can use the linkage stack or the save area its calling program provides. An AR mode program ***must*** use the linkage stack.

Caller-Provided Save Area

A primary mode calling program must provide its target program an 18-word register save area. Likewise, an AR mode program that switches to primary mode and then makes a call must provide a register save area. In both cases, the calling program obtains storage for the save area from its primary address space. The save area must begin on a word boundary. Before invoking the target program, the calling program loads the address of the save area into general purpose register 13.

System-Provided Linkage Stack

The system provides the linkage stack where a target program can save the calling program's access registers and general purpose registers (AR/GPRs). Use of the linkage stack has the following advantages:

- The linkage stack saves both ARs and GPRs; the caller-provided save area saves only GPRs.
- The system provides the linkage stack for use by all programs. The stack eliminates the need for the AR mode calling program to obtain storage for a save area and then pass the address to its target program.
- The save areas are located in one place, rather than chained throughout the user's address space.
- User programs cannot accidentally make changes to the linkage stack.

Using the Linkage Stack

To add an entry to the linkage stack, the target program issues the BAKR instruction. The BAKR instruction stores all GPRs and ARs on the linkage stack. The target program must then indicate that it used the linkage stack, which is useful information for anyone who later needs to trace the program linkages. The procedure for indicating use of the linkage stack is described in:

- "Primary Mode Programs Receiving Control" on page 2-5
- "AR Mode Programs Receiving Control" on page 2-7

When the target program is ready to return to the calling program, it issues the PR instruction. The PR instruction restores the calling program's AR/GPRs 2 - 14, removes the entry from the linkage stack, and returns control to the calling program.

Example of Using the Linkage Stack

In this example, an AR mode target program receives control from another program, either in primary mode or AR mode. The calling program can make the call through the following two instructions:

```
L     15,=V(PGM)
BALR  14,15
```

The target program uses the linkage stack to save the calling program's registers. It uses the STORAGE macro to obtain storage for its own save area. The code is in 31-bit addressing mode and is reentrant.

```
PGM   CSECT
PGM   AMODE  31
PGM   RMODE ANY
      BAKR   14,0                  SAVE CALLER'S ARS AND GPRS
*                                  ON LINKAGE STACK
      SAC    512                   SWITCH TO AR ADDRESSING MODE
      LAE    12,0(15,0)            SET UP PROGRAM BASE REGISTER
*                                  AND ADDRESSING REGISTER
      USING PGM,12
      STORAGE OBTAIN,LENGTH=72     GET MY REENTRANT SAVEAREA
      LAE    13,0(0,1)             PUT MY SAVEAREA ADDRESS IN AR/GPR13
      MVC    4(4,13),=C'F1SA'      PUT ACRONYM INTO MY SAVEAREA BACKWARD
*                                  POINTER INDICATING REGS SAVED ON STACK
*   END OF ENTRY CODE, BEGIN PROGRAM CODE HERE
:
*   BEGIN EXIT CODE
      LAE    1,0(0,13)             COPY MY SAVEAREA ADDRESS
      STORAGE RELEASE,ADDR=(1),LENGTH=72 FREE MY REENTRANT SAVEAREA
      SLR    15,15                 SET RETURN CODE OF ZERO
      PR                           RESTORE CALLER'S ARs AND
*                                  GPRS 2-14 AND RETURN TO CALLER
      END
```

Using a Caller-Provided Save Area

When it receives control, the target program saves the GPRs in the 18-word caller-provided save area pointed to by GPR 13. The format of this area is shown in Figure 2-1. As indicated by this figure, the contents of each GPR, except GPR 13, must be saved in a specific location within the save area. GPR 13 is not saved; it holds the address of the save area.

Word	Contents
0	Used by language products
1	Address of previous save area (stored by calling program)
2	Address of next save area (stored by target program)
3	GPR 14 (return address)
4	GPR 15 (entry address)
5 - 17	GPRs 0 - 12

Figure 2-1. Format of the Save Area

You can save GPRs either with a store-multiple (STM) instruction or with the SAVE macro. Use the following STM instruction to place the contents of all GPRs except GPR 13 in the proper words of the save area:

```
STM 14,12,12(13)
```

The SAVE macro stores GPRs in the save area. Code the GPRs to be saved in the same order as in an STM instruction. The following example of the SAVE macro places the contents of all GPRs except GPR 13 in the proper words of the save area.

```
PROGNAME        SAVE (14,12)
```

Later, the program can use the RETURN macro to restore GPRs and return to the caller.

Whether or not the target program obtains its own save area for another program, it must save the address of the calling program's save area (which it used). If the target program is creating a save area for another program, it:

1. Stores the address of the calling program's save area (the address passed in register 13) in the second word of its own save area.
2. Stores the address of its own save area (the address the target program will pass to another program in register 13) in the third word of the calling program's save area.

These two steps enable the target program to find the save area when it needs it to restore the registers, and they enable a trace from save area to save area should one be necessary while examining a dump.

If the target program is not providing a save area for another program, it can keep the address of the calling program's save area in GPR 13 or store it in a location in virtual storage.

Example of Using the Caller-Provided Save Area

In this example, a primary mode target program receives control in primary mode from either a primary mode or AR mode calling program. The calling program provided an 18-word save area pointed to by GPR 13. The calling program can make the call through the following two instructions:

```
L     15,=V(PGM)
BALR  14,15
```

The target program saves its calling program's registers in the save area that the calling program provides. It uses the GETMAIN macro to obtain storage for its own save area. The code is in 31-bit addressing mode and is reentrant.

```
PGM   CSECT
PGM   AMODE  31
PGM   RMODE ANY
      STM    14,12,12(13)      SAVE CALLER'S REGISTERS IN CALLER-
*                              PROVIDED R13 SAVE AREA
      LR     12,15             SET UP PROGRAM BASE REGISTER
      USING  PGM,12
      GETMAIN RU,LV=72         GET MY REENTRANT SAVEAREA
      ST     13,4(,1)          SAVE CALLER'S SAVEAREA ADDRESS IN MY
*                              SAVEAREA (BACKWARD CHAIN)
      ST     1,8(,13)          SAVE MY SAVEAREA ADDRESS IN CALLER'S
*                              SAVEAREA (FORWARD CHAIN)
      LR     13,1              PUT MY SAVEAREA ADDRESS IN R13
*   END OF ENTRY CODE, BEGIN PROGRAM CODE HERE
:
*   BEGIN EXIT CODE
      LR     1,13              COPY MY SAVEAREA ADDRESS
      L      13,4(,13)         RESTORE CALLER'S SAVEAREA ADDRESS
      FREEMAIN RU,A=(1),LV=72  FREE MY REENTRANT SAVEAREA
      SLR    15,15             SET RETURN CODE OF ZERO
      L      14,12(,13)        RESTORE CALLER'S R14
      LM     2,12,28(13)       RESTORE CALLER'S R2-R12
      BR     14                RETURN TO CALLER
      END
```

Establishing a Base Register

Each program must establish a base register immediately after it saves the calling program's registers. When selecting a base register, keep in mind that:

- Some instructions alter register contents (for example, TRT alters register 2). A complete list of instructions and their processing is available in *Principles of Operation*.
- Registers 13 through 1 are used during program linkage.

Register 12 is generally a good choice for base register.

Linkage Procedures for Primary Mode Programs

A primary mode program can call primary mode programs or AR mode programs. A primary mode program can be called by other primary mode programs or by an AR mode program that has switched to primary mode. The following sections summarize the linkage procedures a primary mode program follows when it receives control, when it returns control to a caller, and when it calls another program.

Primary Mode Programs Receiving Control

When a primary mode program receives control after being called, it can save the calling program's registers on the linkage stack or in the caller-provided save area.

A primary mode program that uses the linkage stack must:

- Issue a BAKR instruction to save the caller's GPRs and ARs on the linkage stack.
- Establish a GPR as a base register.
- Set GPR 13 to indicate that the caller's registers are saved on the linkage stack:
  - If the program intends to call another program, obtain an 18-word save area on a word boundary in the primary address space. Set the second word of

the save area to the character string 'F1SA' and load GPR 13 with the save area address.

- If the program does not intend to call another program, do **one** of following:
  - Obtain an 18-word save area on a word boundary in the primary address space. Set the second word of the save area to the character string 'F1SA' and load the save area address into GPR 13.
  - Load 0 into GPR 13.
  - Set the second word of a two-word area in the primary address space to the character string 'F1SA'. Load the address of the two-word area into GPR 13.

A primary mode program that uses the caller-provided save area must:

- Save GPRs 0 - 12, 14, and 15 in the caller-provided save area pointed to by GPR 13.
- Establish a base register.
- Obtain an 18-word save area on a word boundary in the primary address space.
- Store the address of the caller's save area and the forward and back chains of its own save area, as the comments in "Example of Using the Caller-Provided Save Area" on page 2-4 indicate.

Note that the linkage conventions assume that a primary mode program does not use ARs. By leaving the contents of the ARs untouched, the program preserves the ARs across program linkages.

Primary Mode Programs Returning Control

The method that a primary mode program uses to return control to a caller depends on whether the primary mode program used the linkage stack or the caller-provided save area.

A primary mode program that uses the linkage stack must:

- Place parameter information to return to the caller, if any, into GPR 0, 1, or both. For information about passing information through a parameter list, see "Conventions for Passing Information Through a Parameter List" on page 2-8.
- Load the return code, if any, into GPR 15.
- Issue the PR instruction. The PR instruction restores the caller's AR/GPRs 2 - 14 from the linkage stack, removes the entry from the linkage stack, and returns control to the caller.

A primary mode program that uses the caller-provided save area must:

- Place parameter information to return to the caller, if any, into GPR 0, 1, or both. For information about passing information through a parameter list, see "Conventions for Passing Information Through a Parameter List" on page 2-8.
- Load GPR 13 with the address of the save area that the program passed when it made the call.
- Load the return code, if any, into GPR 15. Otherwise, restore GPR 15 to the value it had when the program was called.
- Restore GPRs 2 - 12 and 14.
- Return to the calling program.

Primary Mode Programs Calling Another Program

When a primary mode program calls another program, the calling program must:

- Place the address of its 18-word save area into GPR 13.
- Load parameter information, if any, into GR 0, GR 1, or both.
- Place the entry point address of the target program into GPR 15.
- Call the target program

Linkage Procedures for AR Mode Programs

An AR mode program can be called by other AR mode programs or by primary mode programs. The following sections summarize the linkage procedures an AR mode program must follow when it receives control, when it returns control to a caller, and when it calls another program.

AR Mode Programs Receiving Control

When an AR mode program receives control, it must:

- Issue a BAKR instruction to save the caller's GPRs and ARs on the linkage stack. (Although a primary mode caller provides a save area, an AR mode target program does not use the area.
- Establish a GPR as a base register and load an ALET of 0 into the corresponding AR. An ALET of 0 causes the system to reference an address within the primary address space.
- Set GPR 13 to indicate that the caller's registers are saved on the linkage stack:
  - If the program intends to switch to primary mode and call another program, obtain an 18-word save area on a word boundary in the primary address space. Set the second word of the save area to the character string 'F1SA' and load GPR 13 with the save area address. Set AR 13 to zero to indicate that the storage resides in the primary address space.
  - If the program does not intend to switch to primary mode and call a program, do **one** of following:
    - Obtain an 18-word save area on a word boundary in the primary address space. Set the second word of the save area to the character string 'F1SA' and load the save area address into GPR 13.
    - Load 0 into GPR 13.
    - Set the second word of a two word area in the primary address space to the character string 'F1SA'. Load the address of the two word area into GPR 13.

AR Mode Programs Returning Control

To return control to the calling program, an AR mode program must:

- Place parameter information to return to the caller, if any, into AR/GPR 0, AR/GPR 1, or both. For information about passing information through a parameter list, see "Conventions for Passing Information Through a Parameter List" on page 2-8.
- Load the return code, if any, into GPR 15.
- Issue the PR instruction. The PR instruction restores the caller's AR/GPRs 2 - 14 from the linkage stack, removes the entry from the linkage stack, and returns control to the caller.

AR Mode Programs Calling Another Program

The definition of an AR mode program, as stated in the beginning of this chapter, includes the fact that such a program might switch from one ASC mode to another. Procedures for an AR mode program calling another program differ depending on whether the AR mode program is in primary mode or AR mode at the time of the call.

To make the call while it is in AR mode, an AR mode program must:

- Load parameter information, if any, into AR/GPR 0, AR/GPR 1, or both. For information about passing information through a parameter list, see "Conventions for Passing Information Through a Parameter List."
- Place the entry point address of the target program into GPR 15. There is no need to load an ALET into AR 15.
- Call the target program

To make the call while it is in primary mode, an AR mode program must follow the linkage conventions described in "Primary Mode Programs Calling Another Program" on page 2-7.

Conventions for Passing Information Through a Parameter List

The ASC mode of a calling program at the time it makes a call determines whether addresses that the program passes are ALET-qualified. The following two sections describe how programs in primary mode and AR mode pass parameters through a parameter list.

Program in Primary Mode

If the calling program is in primary mode, the parameter list must be in the primary address space. All addresses passed by the programs must be contained in the primary address space and must not be ALET-qualified. The program that passes parameter data can use GPRs 0 and 1, or both. To pass the address of a parameter list, the program should use GPR 1.

For a good example of how your primary mode programs can pass parameters, consider the way the system uses a register to pass information in the PARM field of an EXEC statement to your program. When your program receives control from the system, register 1 contains the address of a fullword on a fullword boundary in your program's address space (see Figure 2-2). The high-order bit (bit 0) of this word is set to 1. The system uses this convention to indicate the last word in a variable-length parameter list. Bits 1-31 of the fullword contain the address of a two-byte length field on a halfword boundary. The length field contains a binary count of the number of bytes in the PARM field, which immediately follows the length field. If the PARM field was omitted in the EXEC statement, the count is set to zero. To prevent possible errors, always use the count as a length attribute in acquiring the information in the PARM field.

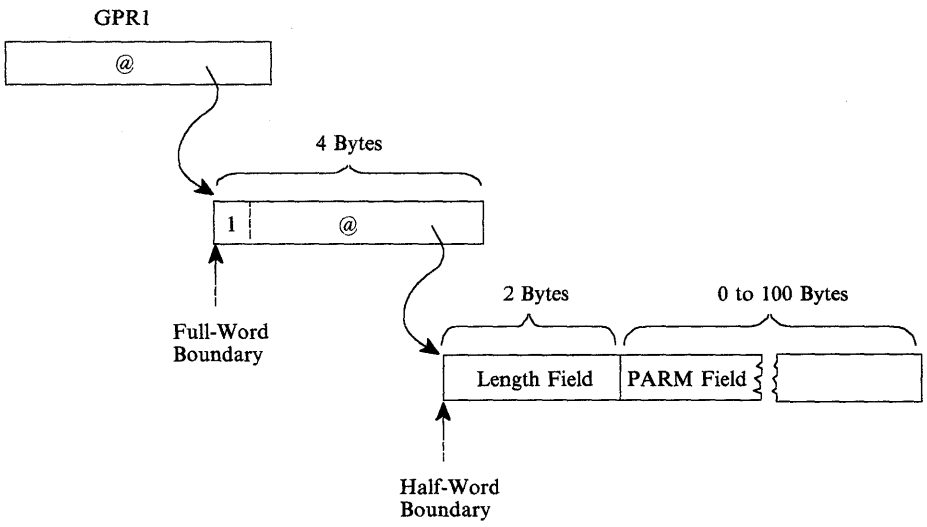


Figure 2-2. Primary Mode Parameter List

Programs in AR Mode

If the calling program is in AR mode, all addresses that it passes, whether they are in a GPR or in a parameter list, must be ALET-qualified. A parameter list can be in an address space other than the calling program's primary address space or in a data space, but it cannot be in the calling program's secondary address space.

Figure 2-3 shows one way to format addresses and ALETs in a parameter list. The addresses passed to the called program are at the beginning of the list and their associated ALETs follow the addresses. Notice that the third address has the high order bit set on to indicate the size of the list.

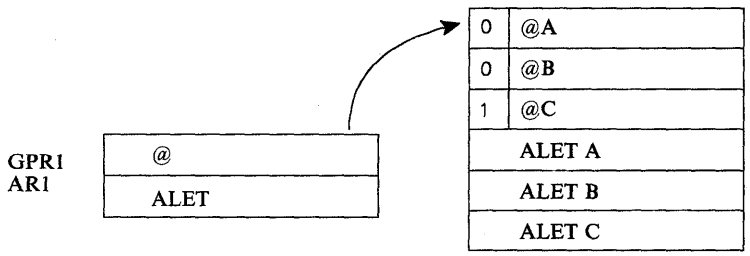


Figure 2-3. AR Mode Parameter List

All addresses that an AR mode target program returns to an AR mode caller, whether the address is in GPR 0 or 1 or in a parameter list, must be ALET-qualified.

Chapter 3. Subtask Creation and Control

The control program creates one task in the address space as a result of initiating execution of the job step (the job step task). You can create additional tasks in your program. However, if you do not, the job step task is the only task in the address space being executed. The benefits of a multiprogramming environment are still available even with only one task in the job step; work is still being performed for other address spaces when your task is waiting for an event, such as an input operation, to occur.

The advantage in creating additional tasks within the job step is that more tasks are competing for control. When a wait condition occurs in one of your tasks, it is not necessarily a task from some other address space that gets control; it may be one of your tasks, a portion of your job.

The general rule is that you should choose parallel execution of a job step (that is, more than one task in an address space) only when a significant amount of overlap between two or more tasks can be achieved. You must take into account the amount of time taken by the control program in establishing and controlling additional tasks, and your increased effort to coordinate the tasks and provide for communications between them.

Creating the Task

A new task is created by issuing an ATTACH, or, if your program runs in access register ASC mode, an ATTACHX macro. The task that is active when the ATTACH or ATTACHX is issued is the originating task; the newly created task is the subtask of the originating task. The subtask competes for control in the same manner as any other task in the system, on the basis of priority (both address space priority and task priority within the address space) and the current ability to use a processor. The address of the task control block for the subtask is returned in register 1.

If the ATTACH or ATTACHX executes successfully, control returns to the user with a return code of 0 in register 15.

The entry point in the load module to be given control when the subtask becomes active is specified as it is in a LINK or LINKX macro, that is, through the use of the EP, EPLOC, and DE parameters. The use of these parameters is discussed in Chapter 4, "Program Management." You can pass parameters to the subtask using the PARAM and VL parameters, also described under the LINK macro. Additional parameters deal with the priority of the subtask, provide for communication between tasks, specify libraries to be used for program linkages, and establish an error recovery environment for the new subtask.

Priorities

This section considers three priorities: address space priorities, task priorities, and subtask priorities.

Address Space Priority

When a job is initiated, the control program creates an address space. All successive steps in the job execute in the same address space. The address space has a dispatching priority, which is normally determined by the control program. The control program will select, and alter, the priority in order to achieve the best load balance in the system, that is, in order to make the most efficient use of processor time and other system resources.

You might want some jobs to execute at a different address space priority than the default priority assigned by the system. To assign a priority, code DPRTY=(value1,value2) on the EXEC statement. The address space priority is then determined as follows:

address space dispatching priority = (value1 x 16) + value2

Once the address space dispatching priority is set, only the control program can change it. (Thus, there is no limit priority associated with an address space.) However, you can set a new address space priority for succeeding job steps by specifying different values in the DPRTY parameter on the EXEC statement.

The IEAIPSxx and IEAICSxx members of SYS1.PARMLIB can override the dispatching priority specified by the DPRTY parameter. The control program assigns the priority obtained from IEAIPSxx to jobsteps that request a dispatching priority falling within specific installation defined limits. IEAICSxx directs jobs into specific performance groups thereby affecting their priority. See *MVS* Initialization and Tuning Guide* for additional information.

Task Priority

Each task in an address space has a limit priority and a dispatching priority associated with it. The control program sets these priorities when a job step is initiated. When you use the ATTACH or ATTACHX macro to create other tasks in the address space, you can use the LPMOD and DPMOD parameters to give them different limit and dispatching priorities.

The dispatching priorities of the tasks in an address space do not affect the order in which the control program selects jobs for execution because that order is selected on the basis of address space dispatching priority. Once the control program selects an address space for dispatching, it selects from within the address space the highest priority task awaiting execution. Thus, task priorities may affect processing within an address space. Note, however, that in a multiprocessing system, task priorities cannot guarantee the order in which the tasks will execute because more than one task may be executing simultaneously in the same address space on different processors. Page faults may alter the order in which the tasks execute.

* MVS is a trademark of the IBM Corporation

Subtask Priority

When a subtask is created, the limit and dispatching priorities of the subtask are the same as the current limit and dispatching priorities of the originating task except when the subtask priorities are modified by the LPMOD and DPMOD parameters of the ATTACH and ATTACHX macro. The LPMOD parameter specifies the signed number to be subtracted from the current limit priority of the originating task. The result of the subtraction is assigned as the limit priority of the subtask. If the result is zero or negative, zero is assigned as the limit priority. The DPMOD parameter specifies the signed number to be added to the current dispatching priority of the originating task. The result of the addition is assigned as the dispatching priority of the subtask, unless the number is greater than the limit priority or less than zero. In that case, the limit priority or 0, respectively, is used as the dispatching priority.

Assigning and Changing Priority

Assign tasks with a large number of I/O operations a higher priority than tasks with little I/O, because the tasks with much I/O will be in a wait condition for a greater amount of time. The lower priority tasks will be executed when the higher priority tasks are in a wait condition. As the I/O operations are completed, the higher priority tasks get control, so that more I/O can be started.

You can change the priorities of subtasks by using the CHAP macro. The CHAP macro changes the dispatching priority of the active task or one of its subtasks by adding a positive or negative value. The dispatching priority of an active task can be made less than the dispatching priority of another task. If this occurs and the other task is dispatchable, it would be given control after execution of the CHAP macro.

You can also use the CHAP macro to increase the limit priority of any of an active task's subtasks. An active task cannot change its own limit priority. The dispatching priority of a subtask can be raised above its own limit priority, but not above the limit of the originating task. When the dispatching priority of a subtask is raised above its own limit priority, the subtask's limit priority is automatically raised to equal its new dispatching priority.

Task and Subtask Communications

The task management information in this section is required only for establishing communications among tasks in the same job step. The relationship of tasks in a job step is shown in Figure 3-1. The horizontal lines in Figure 3-1 separate originating tasks and subtasks; they have no bearing on task priority. Tasks A, A1, A2, A2a, B, B1 and B1a are all subtasks of the job-step task; tasks A1, A2, and A2a are subtasks of task A. Tasks A2a and B1a are the lowest level tasks in the job step. Although task B1 is at the same level as tasks A1 and A2, it is not considered a subtask of task A.

Task A is the originating task for both tasks A1 and A2, and task A2 is the originating task for task A2a. A hierarchy of tasks exists within the job step. Therefore the job step task, task A, and task A2 are predecessors of task A2a, while task B has no direct relationship to task A2a.

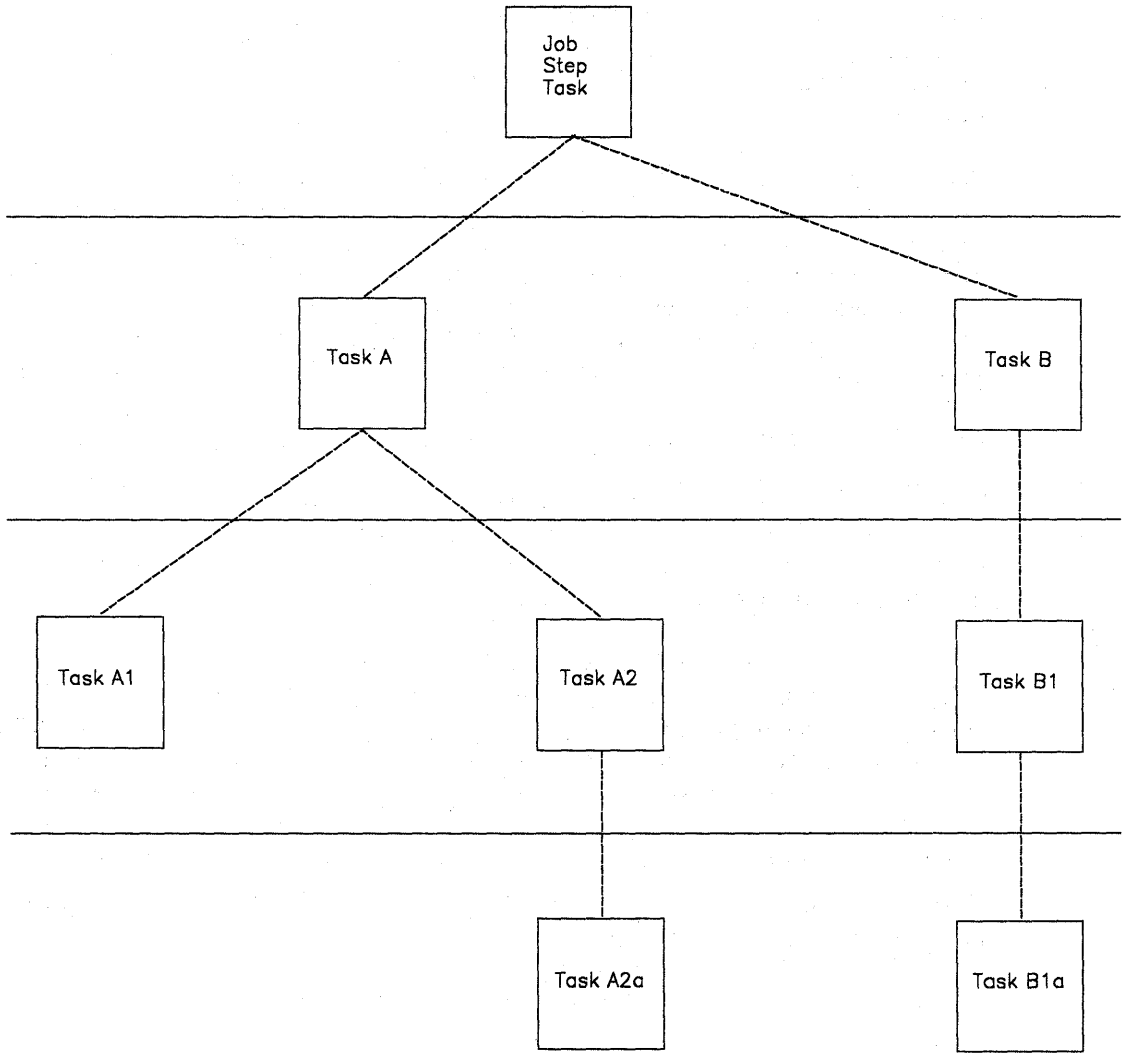


Figure 3-1. Levels of Tasks in a Job Step

All of the tasks in the job step compete independently for processor time; if no constraints are provided, the tasks are performed and are terminated asynchronously. However, since each task is performing a portion of the same job step, some communication and constraints between tasks are required, such as notifying each other when a subtask completes. If a predecessor task task attempts to terminate before all of its subtasks are complete, those subtasks and the predecessor task are abnormally terminated.

Two parameters, the ECB and ETXR parameters, are provided in the ATTACH or ATTACHX macro to assist in communication between a subtask and the originating task. These parameters are used to indicate the normal or abnormal termination of a subtask to the originating task. If you coded the ECB or ETXR parameter, or both, in the ATTACH or ATTACHX macro, the task control block of the subtask is not removed from the system when the subtask is terminated. The originating task must remove the task control block from the system after termination of the subtask by issuing a DETACH. If you specified the ECB parameter in the ATTACH or ATTACHX macro, the ECB must be in storage addressable by the attaching task and control program so that the issuer of ATTACH can wait on it (using the WAIT macro) and the control program can post it on behalf of the terminating task. The task control blocks for all subtasks must be removed before the originating task can terminate normally.

The ETXR parameter specifies the address of an end-of-task exit routine in the originating task, which is to be given control when the subtask being created is terminates. The end-of-task routine is given control asynchronously after the subtask has terminated and must therefore be in virtual storage when it is required. After the control program terminates the subtask, the end-of-task routine specified is scheduled to be executed. It competes for CPU time using the priority of the originating task and of its address space and can receive control even though the originating task is in the wait condition. Although the DETACH does not have to be issued in the end-of-task routine, this is a good place for it.

The ECB parameter specifies the address of an event control block (discussed under "Task Synchronization"), which is posted by the control program when the subtask is terminated. After posting occurs, the event control block contains the completion code specified for the subtask.

If you specified neither the ECB nor the ETXR parameter in the ATTACH or ATTACHX macro, the task control block for the subtask is removed from the system when the subtask terminates. Its originating task does not have to issue a DETACH. A reference to the task control block in a CHAP or a DETACH macro in this case is as risky as task termination. Since the originating task is not notified of subtask termination, you may refer to a task control block that has been removed from the system, which would cause the active task to be abnormally terminated.

Note: The originating task is abended if it attempts to normally terminate when it has active subtasks.

Chapter 4. Program Management

This chapter discusses facilities that will help you to design your programs. It includes descriptions of the residency mode and addressing mode of programs, linkage considerations, load module structures, facilities for passing control between programs, and the use of the associated macro.

Residency and Addressing Mode of Programs

The control program ensures that each load module is loaded above or below 16 megabytes virtual as appropriate and that it is invoked in the correct addressing mode (24-bit or 31-bit). The placement of the module above or below 16 megabytes depends on the residency mode (RMODE) that you define for the module. Whether a module executes in 24-bit or 31-bit addressing mode depends on the addressing mode (AMODE) that you define for the module.

When a program is executing in 24-bit addressing mode, the system treats both instruction and data addresses as 24-bit addresses. This allows programs executing in 24-bit addressing mode to address 16 megabytes (16,777,216 bytes) of storage. Similarly, when a program is executing in 31-bit addressing mode, the system treats both instruction and data addresses as 31-bit addresses. This allows a program executing in 31-bit addressing mode to address 2 gigabytes (2,147,483,648 bytes or 128 x 16 megabytes) of storage.

You can define the residency mode and the addressing mode of a program in the source code. Figure 4-1 shows an example of the definition of the AMODE and RMODE attributes in the source code. This example defines the addressing mode of the load module as 31-bit and the residency mode of the load module as 24-bit. Therefore, the program will receive control in 31-bit addressing mode and will reside below 16 megabytes.

```
SAMPLE CSECT
SAMPLE AMODE  31
SAMPLE RMODE  24
```

Figure 4-1. Assembler Definition of AMODE/RMODE

Version 2 of Assembler H places the AMODE and RMODE in the external symbol dictionary (ESD) of the output object module for use by the linkage editor. The linkage editor passes this information on to the control program through the directory entry for the partitioned data set (PDS) that contains the load module and the composite external symbol dictionary (CESD) record in the load module. You can also specify the AMODE/RMODE attributes of a load module by using linkage editor control cards. Chapter 5, "Understanding 31-Bit Addressing" on page 5-1 contains additional information about residency and addressing mode; *Linkage Editor and Loader* contains information about the linkage editor control cards.

Residency Mode Definitions

The control program uses the RMODE attribute from the PDS directory entry for the module to load the program above or below 16 megabytes. The RMODE attribute can have one of the following values:

24 specifies that the program must reside in 24-bit addressable virtual storage.

ANY specifies that the program can reside anywhere in virtual storage because the code has no virtual storage residency restrictions.

Note: The default value for RMODE is 24.

Addressing Mode Definitions

The AMODE attribute, located in the PDS directory entry for the module, specifies the addressing mode that the module expects at entry. Bit 32 of the program status word (PSW) indicates the addressing mode of the program that is executing. The system supports programs that execute in either 24-bit or 31-bit addressing mode. The AMODE attribute can have one of the following values:

24 specifies that the program is to receive control in 24-bit addressing mode.

31 specifies that the program is to receive control in 31-bit addressing mode.

ANY specifies that the program is to receive control in either 24-bit or 31-bit addressing mode.

Note: The default value for AMODE is 24.

Linkage Considerations

The system supports programs that execute in either 24-bit or 31-bit addressing mode. The following branch instructions take addressing mode into consideration:

Branch and link (BAL)
Branch and link, register form (BALR)
Branch and save (BAS)
Branch and save, register form (BASR)
Branch and set mode (BSM)
Branch and save and set mode (BASSM)
Branch and stack (BAKR)

See *Principles of Operation* for a complete description of how these instructions function. The following paragraphs provide a general description of these branch instructions.

The BAL and BALR instructions are unconditional branch instructions (to the address in operand 2). BAL and BALR function differently depending on the addressing mode in which you are executing. The difference is in the linkage information passed in the link register when these instructions execute. In 31-bit addressing mode, the link register contains the AMODE indicator (bit 0) and the address of the next sequential instruction (bits 1-31); in 24-bit addressing mode, the link register contains the instruction length code, condition code, program mask, and the address of the next sequential instruction.

BAS and BASR perform the same function that BAL and BALR perform when BAL and BALR execute in 31-bit addressing mode.

The BSM instruction provides problem programs with a way to change the AMODE bit in the PSW. BSM is an unconditional branch instruction (to the address in operand 2) that saves the current AMODE in the high-order bit of the link register (operand 1), and sets the AMODE indicator in the PSW to agree with the AMODE of the address to which you are transferring control (that is, the high order bit of operand 2).

The BASSM instruction functions in a manner similar to the BSM instruction. In addition to saving the current AMODE in the link register, setting the PSW AMODE bit, and transferring control, BASSM also saves the address of the next sequential instruction in the link register thereby providing a return address.

BASSM and BSM are used for entry and return linkage in a manner similar to BALR and BR. The major difference from BALR and BR is that BASSM and BSM can save and change addressing mode.

The BAKR instruction is an unconditional branch to the address in operand 2. In addition to the branching action, it adds an entry to the linkage stack.

Passing Control Between Programs with the Same AMODE

If you are passing control between programs that execute in the same addressing mode, there are several combinations of instructions that you can use. Some of these combinations are:

Transfer	Return
BAL/BALR	BR
BAS/BASR	BR

Passing Control Between Programs with Different AMODEs

If you are passing control between programs executing in different addressing modes, you must change the AMODE indicator in the PSW. The BASSM and BSM instructions perform this function for you. You can transfer to a program in another AMODE using a BASSM instruction and then return by means of a BSM instruction. This sequence of instructions ensures that both programs execute in the correct AMODE.

Figure 4-2 shows an example of passing control between programs with different addressing modes. In the example, TEST executes in 24-bit AMODE and EP1 executes in 31-bit AMODE. Before transferring control to EP1, the TEST program loads register 15 with EPA, the pointer defined entry point address (that is, the address of EP1 with the high order bit set to 1 to indicate 31-bit AMODE). This is followed by a BASSM 14,15 instruction, which performs the following functions:

- Sets the high-order bit of the link register (register 14) to 0 (because TEST is currently executing in 24-bit AMODE) and puts the address of the next sequential instruction into bits 1-31.
- Sets the PSW AMODE bit to 1 to agree with bit 0 of register 15.
- Transfers to EP1 (the address in bits 1-31 of register 15).

The EP1 program executes in 31-bit AMODE. Upon completion, EP1 sets a return code in register 15 and executes a BSM 0,14 instruction, which performs the following functions:

- Sets the PSW AMODE bit to 0 to correspond to the high-order bit of register 14.
- Transfers control to the address following the BASSM instruction in the TEST program.

```
TEST    CSECT
TEST    AMODE   24
TEST    RMODE   24
        .
        .
        L       15,EPA      OBTAIN TRANSFER ADDRESS
        BASSM   14,15       SWITCH AMODE AND TRANSFER
        .
        .
        EXTRN   EP1
EPA     DC      A(X'80000000'+EP1) POINTER DEFINED ENTRY POINT ADDRESS
        .
        .
        END
```

```
EP1     CSECT
EP1     AMODE   31
EP1     RMODE   ANY
        .
        .
        SLR     15,15       SET RETURN CODE 0
        BSM     0,14        RETURN TO CALLER'S AMODE AND TRANSFER
        END
```

Figure 4-2. Example of Addressing Mode Switch

Load Module Structure Types

Each load module used during a job step can be designed in one of three load module structures: *simple*, *planned overlay*, or *dynamic*. A simple structure does not pass control to any other load modules during its execution, and comes into into virtual storage all at one time. A planned overlay structure may, if necessary, pass control to other load modules during its execution, and it does not come into virtual storage all at one time. Instead, segments of the load module reuse the same area of virtual storage. A dynamic structure comes into virtual storage all at one time, and passes control to other load modules during its execution. Each of the load modules to which control is passed can be one of the three structure types. Characteristics of the load module structure types are summarized in Figure 4-3.

Because the large capacity of virtual storage eliminates the need for complex overlay structures, planned overlays will not be discussed further.

Structure Type	Loaded All at One Time	Passes Control to Other Load Modules
Simple	Yes	No
Planned Overlay	No	Optional
Dynamic	Yes	Yes

Figure 4-3. Characteristics of Load Modules

Simple Structure

A simple structure consists of a single load module produced by the linkage editor. The single load module contains all of the instructions required and is paged into central storage by the control program as it is executed. The simple structure can be the most efficient of the two structure types because the instructions it uses to pass control do not require control-program assistance. However, you should design your program to make most efficient use of paging.

Dynamic Structure

A dynamic structure requires more than one load module during execution. Each required load module can operate as either a simple structure or another dynamic structure. The advantages of a dynamic structure over a simple structure increase as the program becomes more complex, particularly when the logical path of the program depends on the data being processed. The load modules required in a dynamic structure are paged into central storage when required, and can be deleted from virtual storage when their use is completed.

Load Module Execution

Depending on the configuration of the operating system and the macros used to pass control, execution of the load modules is serial or in parallel. Execution is serial in the operating system unless you use an ATTACH or ATTACHX macro to create a new task. The new task competes for processor time independently with all other tasks in the system. The load module named in the ATTACH or ATTACHX macro is executed in parallel with the load module containing the ATTACH or ATTACHX macro. The execution of the load modules is serial within each task.

The following paragraphs discuss passing control for serial execution of a load module. For information on creating and managing new tasks, see "Creating the Task" on page 3-1.

Passing Control in a Simple Structure

There are certain procedures to follow when passing control to an entry point in the same load module. The established conventions to use when passing control are also discussed. These procedures and conventions are the framework for all program interfaces. Knowledge of the information about addressing contained in the *Assembler Language* publication is required.

Passing Control without Return

Some control sections pass control to another control section of the load module and do not receive control back. An example of this type of control section is a housekeeping routine at the beginning of a program that establishes values, initializes switches, and acquires buffers for the other control sections in the program. Use the following procedures when passing control without return.

Preparing to Pass Control

- Restore the contents of register 14.

  Because control will not be returned to this control section, you must restore the contents of register 14. Register 14 originally contained the address of the location in the calling program (for example, the control program) to which control is to be passed when your program is finished. Since the current control section does not make the return to the calling program, the return address must be passed on to the control section that does make the return.

- Restore the contents of registers 2-12.

  In addition, the contents of registers 2-12 must be unchanged when your program eventually returns control, so you must also restore these registers.

  If control were being passed to the next entry point from the control program, register 15 would contain the entry address. You should use register 15 in the same way, so that the called routine remains independent of the program that passed control to it.

- Use register 1 to pass parameters.

  Establish a parameter list and place the address of the list in register 1. The parameter list should consist of consecutive fullwords starting on a fullword boundary, each fullword containing an *address* to be passed to the called control section. When executing in 24-bit AMODE, each address is located in the three low-order bytes of the word. When executing in 31-bit AMODE, each address is located in bits 1-31 the word. In both addressing modes, set the high-order bit of the last word to 1 to indicate that it is the last word of the list. The system convention is that the list contain addresses only. You may, of course, deviate from this convention; however, when you deviate from any system convention, you restrict the use of your programs to those programmers who know about your special conventions.

- Pass the address of the save area in register 13 just as it was passed to you.

  Since you have reloaded all the necessary registers, the save area that you received on entry is now available, and should be reused by the called control section. By passing the address of the old save area, you save the 72 bytes of virtual storage for a second, unnecessary, save area.

  Note: If you pass a new save area instead of the one received on entry, errors could occur.

Passing Control

- Load register 15 with a V-type address constant for the name of the external entry point, then branch to the address in register 15.

  This is the common way to pass control between one control section and an entry point in the same load module. The external entry point must have been identified using an ENTRY instruction in the called control section if the entry point is not the same as the control section's CSECT name.

Figure 4-4 shows an example of loading registers and passing control. In this example, no new save area is used, so register 13 still contains the address of the old save area. It is also assumed for this example that the control section will pass the same parameters it received to the next entry point. First, register 14 is reloaded with the return address. Next, register 15 is loaded with the address of the external entry point NEXT, using the V-type address constant at the location NEXTADDR. Registers 0-12 are reloaded, and control is passed by a branch instruction using register 15. The control section to which control is passed contains an ENTRY instruction identifying the entry point NEXT.

```
          .
          .
          L   14,12(13)       LOAD CALLER'S RETURN ADDRESS
          L   15,NEXTADDR     ENTRY NEXT
          LM  0,12,20(13)     RETURN CALLER's REGISTERS
          BR  15              NEXT SAVE (14,12)
          .
          .
NEXTADDR  DC  V(NEXT)
```

Figure 4-4. Passing Control in a Simple Structure

Figure 4-5 shows an example of passing a parameter list to an entry point with the same addressing mode. Early in the routine the contents of register 1 (that is, the address of the fullword containing the PARM field address) were stored at the fullword PARMADDR. Register 13 is loaded with the address of the old save area, which had been saved in word 2 of the new save area. The contents of register 14 are restored, and register 15 is loaded with the entry address.

```
          .
          .
          USING *,12          Establish addressability
EARLY     ST    1,PARMADDR    Save parameter address
          .
          .
          L     13,4(13)      Reload address of old save area
          L     0,20(13)
          L     14,12(13)     Load return address
          L     15,NEXTADDR   Load address of next entry point
          LA    1,PARMLIST    Load address of parameter list
          OI    PARMADDR,X'80' Turn on last parameter indicator
          LM    2,12,28(13)   Reload remaining registers
          BR    15            Pass control
          .
          .
PARMLIST  DS    0A
DCBADDRS  DC    A(INDCB)
          DC    A(OUTDCB)
PARMADDR  DC    A(0)
NEXTADDR  DC    V(NEXT)
```

Figure 4-5. Passing Control With a Parameter List

The address of the list of parameters is loaded into register 1. These parameters include the addresses of two data control blocks (DCBs) and the original register 1 contents. The high-order bit in the last address parameter (PARMADDR) is set to 1

using an OR-immediate instruction. The contents of registers 2-12 are restored. (Since one of these registers was the base register, restoring the registers earlier would have made the parameter list unaddressable.) A branch register instruction using register 15 passes control to entry point NEXT.

Passing Control with Return

The control program passed control to your program, and your program will return control when it is through processing. Similarly, control sections within your program will pass control to other control sections, and expect to receive control back. An example of this type of control section is a monitoring routine; the monitor determines the order of execution of other control sections based on the type of input data. Use the following procedures when passing control with return.

Preparing to Pass Control

- Use registers 15 and 1 in the same manner they are used to pass control without return.

  Register 15 contains the entry address in the new control section and register 1 is used to pass a parameter list.

- Ensure that register 14 contains the address of the location to which control is to be returned when the called control section completes execution.

  The address can be the instruction following the instruction which causes control to pass, or it can be another location within the current control section designed to handle all returns.

  Registers 2-12 are not involved in the passing of control; the called control section should not depend on the contents of these registers in any way.

- Provide a new save area for use by the called control section as previously described, and pass the address of that save area in register 13.

  Note that the same save area can be reused after control is returned by the called control section. One new save area is ordinarily all you will require regardless of the number of control sections called.

Passing Control

You may use two standard methods for passing control to another control section and providing for return of control. One is an extension of the method used to pass control without a return, and requires a V-type address constant and a branch, a branch and link, or a branch and save instruction provided both programs execute in the same addressing mode. If the addressing mode changes, use a branch and save and set mode instruction. The other method uses the CALL macro to provide a parameter list and establish the entry and return addresses. With either method, you must identify the entry point by an ENTRY instruction in the called control section if the entry name is not the same as the control section CSECT name. Figure 4-6 and Figure 4-7 illustrate the two methods of passing control; in each example, assume that register 13 already contains the address of a new save area.

Figure 4-6 also shows the use of an inline parameter list and an answer area. The address of the external entry point is loaded into register 15 in the usual manner. A branch and link instruction is then used to branch around the parameter list and to load register 1 with the address of the parameter list. An inline parameter list, such as the one shown in Figure 4-6, is convenient when you are debugging because the parameters involved are located in the listing (or the dump) at the point they are used, instead of at the end of the listing or dump. Note that the high-order bit of the last address parameter (ANSWERAD) is set to 1 to indicate the end of the list. The

area pointed to by the address in the ANSWERAD parameter is an area to be used by the called control section to pass parameters back to the calling control section. This is a possible method to use when a called control section must pass parameters back to the calling control section. Parameters are passed back in this manner so that no additional registers are involved. The area used in this example is twelve words. The size of the area for any specific application depends on the requirements of the two control sections involved.

```
          .
          .
          L       15,NEXTADDR          Entry address in register 15
          CNOP    0,4
          BAL     1,GOOUT              Parameter list address in register 1
PARMLIST  DS      0A                   Start of parameter list
DCBADDRS  DC      A(INDCB)             Input DCB address
          DC      A(OUTDCB)            Output DCB address
ANSWERAD  DC      A(AREA+X'80000000')  Answer area address with
                                       high-order bit on
NEXTADDR  DC      V(NEXT)              Address of entry point
GOOUT     BALR    14,15                Pass control; register 14 contains
                                       return address and current AMODE
RETURNPT  ...
AREA      DC      12F'0'               Answer area from NEXT
```

Note: This example assumes that you are passing control to a program that executes in the same addressing mode as your program. See "Linkage Considerations" on page 4-2 for information on how to handle branches between programs that execute in different addressing modes.

Figure 4-6. Passing Control With Return

```
          CALL    NEXT,(INDCB,OUTDCB,AREA),VL
RETURNPT  ...
AREA      DC      12F'0'
```

Note: You cannot use the CALL macro to pass control to a program that executes in a different addressing mode.

Figure 4-7. Passing Control With CALL

The CALL macro in Figure 4-7 provides the same functions as the instructions in Figure 4-6. When the CALL macro is expanded, the parameters cause the following results:

NEXT - A V-type address constant is created for NEXT, and the address is loaded into register 15.

(INDCB,OUTDCB,AREA) - A-type address constants are created for the three parameters coded within parentheses, and the address of the first A-type address constant is placed in register 1.

VL - The high-order bit of the last A-type address constant is set to 1.

Control is passed to NEXT using a branch and link instruction. The address of the instruction following the CALL macro is loaded into register 14 before control is passed.

In addition to the results described above, the V-type address constant generated by the CALL macro requires the load module with the entry point NEXT to be link edited into the same load module as the control section containing the CALL macro. The *Linkage Editor and Loader* publication tells more about this service.

The parameter list constructed from the CALL macro in Figure 4-7, contains only A-type address constants. A variation on this type of parameter list results from the following coding:

```
CALL   NEXT,(INDCB,(6),(7)),VL
```

In the above CALL macro, two of the parameters to be passed are coded as registers rather than symbolic addresses. The expansion of this macro again results in a three-word parameter list; in this example, however, the expansion also contains instructions that store the contents of registers 6 and 7 in the second and third words, respectively, of the parameter list. The high-order bit in the third word is set to 1 after register 7 is stored. You can specify as many address parameters as you need, and you can use symbolic addresses or register contents as you see fit.

Analyzing the Return

When the control program returns control to a caller after it invokes a system service, the contents of registers 2-13 are unchanged. When control is returned to your control section from the called control section, registers 2-14 contain the same information they contained when control was passed, as long as system conventions are followed. The called control section has no obligation to restore registers 0 and 1; so the contents of these registers may or may not have been changed.

When control is returned, register 15 can contain a return code indicating the results of the processing done by the called control section. If used, the return code should be a multiple of four, so a branching table can be used easily, and a return code of zero should be used to indicate a normal return. The control program frequently uses this method to indicate the results of the requests you make using system macros; an example of the type of return codes the control program provides is shown in the description of the IDENTIFY macro.

The meaning of each of the codes to be returned must be agreed upon in advance. In some cases, either a "good" or "bad" indication (zero or nonzero) will be sufficient for you to decide your next action. If this is true, the coding in Figure 4-8 could be used to analyze the results. Many times, however, the results and the alternatives are more complicated, and a branching table, such as shown in Figure 4-9 could be used to pass control to the proper routine.

Note: Explicit tests are required to ensure that the return code value does not exceed the branch table size.

```
RETURNPT    LTR    15,15    Test return code for zero
            BNZ    ERRORTN  Branch if not zero to error routine
            .
            .
```

Figure 4-8. Test for Normal Return

```
RETURNPT    B   RETTAB(15)  Branch to table using return code
RETTAB      B   NORMAL      Branch to normal routine
            B   COND1       Branch to routine for condition 1
            B   COND2       Branch to routine for condition 2
            B   GIVEUP      Branch to routine to handle impossible situations.
            .
            .
```

Figure 4-9. Return Code Test Using Branching Table

How Control is Returned

In the discussion of the return under "Analyzing the Return" on page 4-10, it was indicated that the control section returning control must restore the contents of registers 2-14. Because these are the same registers reloaded when control is passed without a return, refer to the discussion under "Passing Control without Return" for detailed information and examples. The contents of registers 0 and 1 do not have to be restored.

Register 15 can contain a return code when control is returned. As indicated previously, a return code should be a multiple of four with a return code of zero indicating a normal return. The return codes other than zero that you use can have any meaning, as long as the control section receiving the return codes is aware of that meaning.

The return address is the address originally passed in register 14; you should always return control to that address. If an addressing mode switch is not involved, you can either use a branch instruction such as BR 14, or you can use the RETURN macro. An example of each of these methods of returning control is discussed in the following paragraphs. If an addressing mode switch is involved, you can use a BSM 0,14 instruction to return control. See Figure 4-2 for an example that uses the BSM instruction to return control.

Figure 4-10 shows a portion of a control section used to analyze input data cards and to check for an out-of-tolerance condition. Each time an out-of-tolerance condition is found, in addition to some corrective action, one is added to the one-byte value at the address STATUSBY. After the last data card is analyzed, this control section returns to the calling control section, which bases its next action on the number of out-of-tolerance conditions encountered. The coding shown in Figure 4-10 loads register 14 with the return address. The contents of register 15 are set to zero, and the value at the address STATUSBY (the number of errors) is placed in the low-order eight bits of the register. The contents of register 15 are shifted to the left two places to make the value a multiple of four. Registers 2-12 are reloaded, and control is returned to the address in register 14.

```
         .
         .
         L     13,4(13)      Load address of previous save area
         L     14,12(13)     Load return address
         SR    15,15         Set register 15 to zero
         IC    15,STATUSBY   Load number of errors
         SLA   15,2          Set return code to multiple of 4
         LM    2,12,28(13)   Reload registers 2-12
         BR    14            Return
         .
         .
STATUSBY DC    X'00'
```

Note: This example assumes that you are returning to a program with the same AMODE. If not, use the BSM instruction to transfer control.

Figure 4-10. Establishing a Return Code

The RETURN macro saves coding time. The expansion of the RETURN macro provides instructions that restore a designated range of registers, load a return code in register 15, and branch to the address in register 14. If T is specified, the RETURN macro flags the save area used by the returning control section (that is, the save area supplied by the calling routine). It does this by setting the low-order bit of word four of the save area to one after the registers have been restored. The flag indicates that the control section that used the save area has returned to the calling control section. The flag is useful when tracing the flow of your program in a dump. For a complete record of program flow, a separate save area must be provided by each control section each time control is passed.

You must restore the contents of register 13 before issuing the RETURN macro. Code the registers to be reloaded in the same order as they would have been designated for a load-multiple (LM) instruction. You can load register 15 with the return code before you write the RETURN macro, you can specify the return code in the RETURN macro, or you can reload register 15 from the save area.

The coding shown in Figure 4-11 provides the same result as the coding shown in Figure 4-10. Registers 13 and 14 are reloaded, and the return code is loaded in register 15. The RETURN macro reloads registers 2-12 and passes control to the address in register 14. The save area used is not flagged. The RC=(15) parameter indicates that register 15 already contains the return code, and the contents of register 15 are not to be altered.

```
         .
         .
         L        13,4(13)        Restore save area address
         L        14,12(13)       Return address in register 14
         SR       15,15           Zero register 15
         IC       15,STATUSBY     Load number of errors
         SLA      15,2            Set return code to multiple of 4
         RETURN   (2,12),RC=(15)  Reload registers and return
         .
         .
STATUSBY DC       X'00'
```

Note: You cannot use the RETURN macro to pass control to a program that executes in a different addressing mode.

Figure 4-11. Using the RETURN Macro

Figure 4-12 illustrates another use of the RETURN macro. The correct save area address is again established, and then the RETURN macro is issued. In this example, registers 14 and 0-12 are reloaded, a return code of 8 is placed in register 15, the save area is flagged, and control is returned. Specifying a return code overrides the request to restore register 15 even though register 15 is within the designated range of registers.

```
         .
         .
         L        13,4(13)
         RETURN   (14,12),T,RC=8
```

Figure 4-12. RETURN Macro with Flag

Return to the Control Program

The discussion in the preceding paragraphs has covered passing control within one load module, and has been based on the assumption that the load module was brought into virtual storage because of the program name specified in the EXEC statement. The control program established only one task to be performed for the job step. When the logical end of the program is reached, control passes to the return address passed (in register 14) to the first control section in the control program. When the control program receives control at this point, it terminates the task it created for the job step, compares the return code in register 15 with any COND values specified on the JOB and EXEC statements, and determines whether or not subsequent job steps, if any are present, should be executed.

When your program returns to the control program, your program should use a return code between 0 and 4095 (X"FFF"). A return code of more than 4095 might make return code testing, message processing, and report generation inaccurate.

Passing Control in a Dynamic Structure

The discussion of passing control in a simple structure provides the background for the discussion of passing control in a dynamic structure. Within each load module, control should be passed as in a simple structure. If you can determine which control sections will make up a load module before you code the control sections, you should pass control within the load module without involving the control program. The macros discussed in this section provide increased linkage capability, but they require control program assistance and possibly increased execution time.

Bringing the Load Module into Virtual Storage

The control program automatically brings the load module containing the entry name you specified on the EXEC statement into virtual storage. The control program places the load module above or below 16 megabytes according to its RMODE attribute. Any other load modules you require during your job step are brought into virtual storage by the control program when requested. Make these requests by using the LOAD, LINK, LINKX, ATTACH, ATTACHX, XCTL, and XCTLX macros. The LOAD macro sets the high-order bit of the entry point address to indicate the addressing mode of the load module. The ATTACH, ATTACHX, LINK, LINKX, XCTL, and XCTLX macros use this information to set the addressing mode for the module that gets control. If the AMODE is ANY, the module will get control in the same addressing mode as the program that issued the ATTACH, ATTACHX, LINK, LINKX, XCTL, or XCTLX macro. If a copy of the load module must be brought into storage, the control program places the load module above or below 16 megabytes according to its RMODE attribute. The following paragraphs discuss the proper use of these macro.

Location of the Load Module

Initially, each load module that you can obtain dynamically is located in a library (partitioned data set). This library is the link library, the job or step library, the task library, or a private library.

- The link library (defined by the LNKLSTxx member of SYS1.PARMLIB) is always present and is available to all job steps of all jobs. The control program provides the data control block for the library and logically connects the library to your program, making the members of the library available to your program. For more information, see *MVS Initialization and Tuning Guide*.
- The job and step libraries are explicitly established by including //JOBLIB and //STEPLIB DD statements in the input stream. The //JOBLIB DD statement is placed immediately after the JOB statement, while the //STEPLIB DD statement is placed among the DD statements for a particular job step. The job library is available to all steps of your job, except those that have step libraries. A step library is available to a single job step; if there is a job library, the step library replaces the job library for the step. For either the job library or the step library, the control program provides the data control block and issues the OPEN macro to logically connect the library to your program.

  Authorization: If an authorized program (supervisor state, APF-authorized, PSW key 0 - 7, or PKM 0 - 7) invokes an unauthorized program, the unauthorized program must reside in an APF-authorized library. APF (authorized program facility) prevents authorized programs from accessing any load module that is not in an APF-authorized library. If an authorized program tries to access a module in an APF-authorized library, the system searches for a copy of the module in those libraries. If it finds one, it continues processing with that copy.

If it does not find one, it abends with code 306, even though you might have included a //STEPLIB DD statement for the program. For information about APF-authorization, see the application development books that are available to the programmers that use authorized macros.

- Unique task libraries can be established by using the TASKLIB parameter of the ATTACH or ATTACHX macro. The issuer of the ATTACH or ATTACHX macro is responsible for providing the DD statement and opening the data set or sets. If the TASKLIB parameter is omitted, the task library of the attaching task is propagated to the attached task. In the following example, task A's job library is LIB1. Task A attaches task B, specifying TASKLIB=LIB2 in the ATTACH or ATTACHX macro. Task B's task library is therefore LIB2.
  When task B attaches task C, LIB2 is searched for task C before LIB1 or the link library. Because task B did not specify a unique task library for task C, its own task library (LIB2) is propagated to task C and is the first library searched when task C requests that a module be brought into virtual storage.

  ```
  Task A    ATTACH EP=B,TASKLIB=LIB2
  Task B    ATTACH EP=C
  ```

- Including a DD statement in the input stream defines a private library that is available only to the job step in which it is defined. You must provide the data control block and issue the OPEN macro for each data set. You may use more than one private library by including more than one DD statement and an associated data control block.

A library can be a single partitioned data set, or a collection of such data sets. When it is a collection, you define each data set by a separate DD statement, but you assign a name only to the statement that defines the first data set. Thus, a job library consisting of three partitioned data sets would be defined as follows:

```
//JOBLIB   DD  DSNAME=PDS1,...
//         DD  DSNAME=PDS2,...
//         DD  DSNAME=PDS3...
```

The three data sets (PDS1, PDS2, PDS3) are processed as one, and are said to be *concatenated*. Concatenation and the use of partitioned data sets is discussed in more detail in *Managing Non-VSAM Data Sets*.

Some of the load modules from the link library may already be in virtual storage in an area called the link pack area. The contents of these areas are determined during the nucleus initialization process and will vary depending on the requirements of your installation. The link pack area contains all reenterable load modules from the LPA library, along with installation selected modules from the SVC and link libraries. These load modules can be used by any job step in any job.

With the exception of those load modules contained in this area, copies of all of the reenterable load modules you request are brought into your area of virtual storage and are available to any task in your job step. The portion of your area containing the copies of the load modules is called the job pack area.

The Search for the Load Module

In response to your request for a copy of a load module, the control program searches the job pack area, the task's load list, and the link pack area. If a copy of the load module is found in one of the pack areas, the control program determines whether that copy can be used (see "Using an Existing Copy"). If an existing copy can be used, the search stops. If it cannot be used, the search continues until the

module is located in a library. The load module is then brought into the job pack area or the load list area.

The order in which the control program searches the libraries and pack areas depends on the parameters used in the macro (LINK, LINKX, LOAD, XCTL, XCTLX, ATTACH or ATTACHX) requesting the load module. The parameters that define the order of the search are EP, EPLOC, DE, DCB, and TASKLIB.

Use the TASKLIB parameter only for ATTACH or ATTACHX. You should choose the parameters for the macro that provide the shortest search time. The search of a library actually involves the search of a directory, followed by copying the directory entry into virtual storage, followed by loading the load module into virtual storage. If you know the location of the load module, you should use parameters that eliminate as many of these searches as possible, as indicated in Figure 4-13, Figure 4-14, and Figure 4-15.

The EP, EPLOC, or DE parameter specifies the name of the entry point in the load module. Code one of the three every time you use a LINK, LINKX, LOAD, XCTL, XCTLX, ATTACH, or ATTACHX macro. The optional DCB parameter indicates the address of the data control block for the library containing the load module. Omitting the DCB parameter or using the DCB parameter with an address of zero specifies the data control block for the task libraries, the job or step library, or the link library. If you specified TASKLIB and if the DCB parameter contains the address of the data control block for the link library, the control program searches no other library.

To avoid using "system copies" of modules resident in LPA and LINKLIB, you can specifically limit the search for the load module to the job pack area and the first library on the normal search sequence by specifying the LSEARCH parameter on the LINK, LOAD, or XCTL macro with the DCB for the library to be used.

The following paragraphs discuss the order of the search when the entry name used is a member name.

The EP and EPLOC parameters require the least effort on your part; you provide only the entry name, and the control program searches for a load module having that entry name. Figure 4-13 shows the order of the search when EP or EPLOC is coded, and the DCB parameter is omitted or DCB=0 is coded.

The control program searches:

The job pack area for an available copy.
The requesting task's task library and all the unique task libraries of its preceding tasks.
(Note: For the ATTACH or ATTACHX macro, the attached task's library and all the unique task libraries of its preceding tasks are searched.)
The step library; if there is no step library, the job library (if any).
The link pack area.
The link library.

Figure 4-13. Search for Module, EP or EPLOC Parameter With DCB=0 or DCB Parameter Omitted

When used without the DCB parameter, the EP and EPLOC parameters provide the easiest method of requesting a load module from the link, job, or step library. The control program searches the task libraries before the job or step library, beginning with the task library of the task that issued the request and continuing through the

task libraries of all its antecedent tasks. It then searches the job or step library, followed by the link library.

A job, step, or link library or a data set in one of these libraries can be used to hold one version of a load module, while another can be used to hold another version with the same entry name. If one version is in the link library, you can ensure that the other will be found first by including it in the job or step library. However, if both versions are in the job or step library, you must define the data set that contains the version you want to use before the data set that contains the other version. For example, if the wanted version is in PDS1 and the unwanted version is in PDS2, a step library consisting of these data sets should be defined as follows:

```
//STEPLIB   DD  DSNAME=PDS1,...
//          DD  DSNAME=PDS2,...
```

If, however, the first version of a nonreusable module in the job or step library has been previously loaded and the version in the link library or the second version in the job library is desired, you must code the DCB parameter in the macro.

Use extreme caution when specifying module names in unique task libraries, because duplicate names may cause the wrong module to be brought into virtual storage when a task requests it. Once a module has been loaded from a task library, the module name is known to all tasks in the address space and a copy of that module is given to all tasks requesting that that module name be loaded, regardless of the requester's task library.

If you know that the load module you are requesting is a member of one of the private libraries, you can still use the EP or EPLOC parameter, this time in conjunction with the DCB parameter. Specify the address of the data control block for the private library in the DCB parameter. The order of the search for EP or EPLOC with the DCB parameter is shown in Figure 4-14.

The control program searches:

The job pack area for an available copy.
The specified library.
The link pack area.
The link library.

Figure 4-14. Search for Module, EP or EPLOC Parameters With DCB Parameter Specifying Private Library

Searching a job or step library slows the retrieval of load modules from the link library; to speed this retrieval, you should limit the size of the job and step libraries. You can best do this by eliminating the job library altogether and providing step libraries where required. You can limit each step library to the data sets required by a single step. Some steps (such as compilation) do not require a step library and therefore do not require searching and retrieving modules from the link library. For maximum efficiency, you should define a job library only when a step library would be required for every step, and every step library would be the same.

The DE parameter requires more work than the EP and EPLOC parameters, but it can reduce the amount of time spent searching for a load module. Before you can use this parameter, you must use the BLDL macro to obtain the directory entry for the module. The directory entry is part of the library that contains the module.

To save time, the BLDL macro must obtain directory entries for more than one entry name. Specify the names of the load modules and the address of the data control block for the library when using the BLDL macro; the control program places a copy of the directory entry for each entry name requested in a designated location in virtual storage. If you specify the link library and the job or step library, the directory information indicates from which library the directory entry was taken. The directory entry always indicates the relative track and block location of the load module in the library. If the load module is not located on the library you indicate, a return code is given. You can then issue another BLDL macro specifying a different library.

To use the DE parameter, provide the address of the directory entry and code or omit the DCB parameter to indicate the same library specified in the BLDL macro. The task using the DE parameter should be the same as the one which issued the BLDL or one which has the same job, step, and task library structure as the task issuing the BLDL. The order of the search when the DE parameter is used is shown in Figure 4-15 for the link, job, step, and private libraries.

The preceding discussion of the search is based on the premise that the entry name you specified is the member name. The control program checks for an alias entry point name when the load module is found in a library. If the name is an alias, the control program obtains the corresponding member name from the library directory, and then searches to determine if a usable copy of the load module exists in the job pack area. If a usable copy does not exist in a pack area, a new copy is brought into the job pack area. Otherwise, the existing copy is used, conserving virtual storage and eliminating the loading time.

Directory Entry Indicates Link Library and DCB=0 or DCB Parameter Omitted.
  The job pack area is searched for an available copy.
  The link pack area is searched.
  The module is obtained from the link library.

Directory Entry Indicates Job, Step, or Task Library and DCB=0 or DCB Parameter Omitted.
  The job pack area is searched for an available copy.
  The module is obtained from the task library designated by the 'Z' byte of the DE operand.

DCB Parameter Indicates Private Library
  The job pack area is searched for an available copy.
  The module is obtained from the specified private library.

Figure 4-15. Search for Module Using DE Parameter

As the discussion of the search indicates, you should choose the parameters for the macro that provide the shortest search time. The search of a library actually involves a search of the directory, followed by copying the directory entry into virtual storage, followed by loading the load module into virtual storage. If you know the location of the load module, you should use the parameters that eliminate as many of these unnecessary searches as possible, as indicated in Figure 4-13, Figure 4-14, and Figure 4-15. Examples of the use of these figures are shown in the following discussion of passing control.

Using an Existing Copy

The control program uses a copy of the load module already in the job pack area if the copy can be used. Whether the copy can be used or not depends on the reusability and current status of the load module, that is, the load module attributes, as designated using linkage editor control statements, and whether the load module has already been used or is in use. The status information is available to the control program only when you specify the load module entry name on an EXEC

statement, or when you use ATTACH, ATTACHX, LINK, LINKX, XCTL, or XCTLX macros to transfer control to the load module. The control program protects you from obtaining an unusable copy of a load module if you always "formally" request a copy using these macros (or the EXEC statement). If you pass control in any other manner (for instance, a branch or a CALL macro), the control program, because it is not informed, cannot protect your copy. If your program is in AR mode, and the SYSSTATE ASCENV=AR macro has been issued, use the ATTACHX, LINKX, and XCTLX macros instead of ATTACH, LINK, and XCTL. The macros whose names end with "X" generate code and addresses that are appropriate for AR mode.

All reenterable modules (modules designated as reenterable using the linkage editor) from any library are completely reusable. Only one copy is ever placed in the link pack area or brought into your job pack area, and you get immediate control of the load module. If the module is serially reusable, only one copy is ever placed in the job pack area; this copy is always used for a LOAD macro. If the copy is in use, however, and the request is made using a LINK, LINKX, ATTACH, ATTACHX, XCTL, or XCTLX macro, the task requiring the load module is placed in a wait condition until the copy is available. You should not issue a LINK or LINKX macro for a serially reusable load module currently in use for the same task; the task will be abnormally terminated. (This could occur if an exit routine issued a LINK or LINKX macro for a load module in use by the main program.)

If the load module is not reusable, a LOAD macro will always bring in a new copy of the load module; an existing copy is used only if you issued a LINK, LINKX, ATTACH, ATTACHX XCTL or XCTLX macro and the copy has not been used previously. Remember, the control program can determine if a load module has been used or is in use only if all of your requests are made using LINK, LINKX, ATTACH, ATTACHX, XCTL or XCTLX macros.

Using the LOAD Macro

If a copy of the specified load module is not already in the link pack area, use the LOAD macro to place a copy in the address space. When you issue a LOAD macro, the control program searches for the load module as discussed previously and brings a copy of the load module into the address space if required. When the control program returns control, register 0 contains the addressing mode and the virtual storage address of the entry point specified for the requested load module, and register 1 contains the length of the loaded module (in doublewords) and the authorization code in the high byte. Normally, you should use the LOAD macro only for a reenterable or serially reusable load module, because the load module is retained even though it is not in use.

The control program places the copy of the load module in subpool 251, unless the following three conditions are true:

- The module is reentrant
- The library is authorized
- You are not running under TSO test.

In this case, the control program places the module in subpool 252. Subpool 251 is fetch protected and has a storage key equal to your PSW key. Subpool 252 is not fetch protected and has storage key 0.

The responsibility count for the copy is lowered by one when you issue a DELETE macro during the task which was active when the LOAD macro was issued. When a task is terminated, the count is lowered by the number of LOAD macros issued for the copy when the task was active minus the number of deletions. When the use

count for a copy in a job pack area reaches zero, the virtual storage area containing the copy is made available.

Passing Control with Return

Use the LINK or LINKX macro to pass control between load modules and to provide for return of control. You can also pass control using branch, branch and link, branch and save, or branch and save and set mode instructions or the CALL macro. However, when you pass control in this manner, you must protect against multiple uses of nonreusable or serially reusable modules. You must also be careful to enter the routine in the proper addressing mode. The following paragraphs discuss the requirements for passing control with return in each case.

Using the LINK or LINKX Macro

When you use the LINK or LINKX macro, you are requesting the system to assist you in passing control to another load module. There is some similarity between passing control using a LINK or LINKX macro and passing control using a CALL macro in a simple structure. These similarities are discussed first.

The convention regarding registers 2-12 still applies; the control program does not change the contents of these registers, and the called load module should restore them before control is returned. Unless you are an AR mode program calling an AR mode program that uses the linkage stack, you must provide the address in register 13 of the save area for use by the called load module; the system does not use this save area. You can pass address parameters in a parameter list to the load module using register 1. The LINK or LINKX macro provides the same facility for constructing this list as the CALL macro. Register 0 is used by the control program and the contents may be modified. In certain cases, the contents of register 1 may be altered by the LINK or LINKX macro.

There is also some difference between passing control using a LINK or LINKX macro and passing control using a CALL macro. When you pass control in a simple structure, register 15 contains the entry address and register 14 contains the return address. When the called load module gets control, that is still what registers 14 and 15 contain, but when you use the LINK or LINKX macro, it is the control program that establishes these addresses. When you code the LINK or LINKX macro, you provide the entry name and possibly some library information using the EP, EPLOC, or DE, and DCB parameters, but you have to get this entry name and library information to the control program. The expansion of the LINK or LINKX macro does this by creating a control program parameter list (the information required by the control program) and passing its address to the control program. After the control program finds the entry name, it places the address in register 15.

The return address in your control section is always the instruction following the LINK or LINKX; that is not, however, the address that the called load module receives in register 14. The control program saves the address of the location in your program in its own save area, and places in register 14 the address of a routine within the control program that will receive control. Because control was passed using the control program, return must also be made using the control program. The control program also handles all switching of addressing mode when processing the LINK or LINKX macro.

The control program establishes a use count for a load module when control is passed using the LINK or LINKX macro. This is a separate use count from the count established for LOAD macros, but it is used in the same manner. The count is increased by one when a LINK or LINKX macro is issued and decreased by one

when return is made to the control program or when the called load module issues an XCTL or XCTLX macro.

Figure 4-16 and Figure 4-17 show the coding of a LINK or LINKX macro used to pass control to an entry point in a load module. In Figure 4-16, the load module is from the link, job, or step library; in Figure 4-17, the module is from a private library. Except for the method used to pass control, this example is similar to Figures 10 and 11. A problem program parameter list containing the addresses INDCB, OUTDCB, and AREA is passed to the called load module; the return point is the instruction following the LINK or LINKX macro. A V-type address constant is not generated, because the load module containing the entry point NEXT is not to be edited into the calling load module. Note that the EP parameter is chosen, since the search begins with the job pack area and the appropriate library as shown in Figure 4-13.

```
          LINK    EP=NEXT,PARAM=(INDCB,OUTDCB,AREA),VL=1
RETURNPT  ...
AREA      DC      12F'0'
```

Figure 4-16. Use of the LINK Macro with the Job or Link Library

```
          OPEN    (PVTLIB)
          .
          .
          LINK    EP=NEXT,DCB=PVTLIB,PARAM=(INDCB,OUTDCB,AREA),VL=1
          .
          .
PVTLIB    DCB     DDNAME=PVTLIBDD,DSORG=PO,MACRF=(R)
```

Figure 4-17. Use of the LINK Macro with a Private Library

Figure 4-18 and Figure 4-19 show the use of the BLDL and LINK macros to pass control. Assuming that control is to be passed to an entry point in a load module from the link library, a BLDL macro is issued to bring the directory entry for the member into virtual storage. (Remember, however, that time is saved only if more than one directory entry is requested in a BLDL macro. Only one is requested here for simplicity.)

```
          BLDL    0,LISTADDR
          .
          .
          DS      0H          List description field:
LISTADDR  DC      H'01'           Number of list entries
          DC      H'60'           Length of each entry
NAMEADDR  DC      CL8'NEXT'   Member name
          DS      26H         Area required for directory information
```

Figure 4-18. Use of the BLDL Macro

The first parameter of the BLDL macro is a zero, which indicates that the directory entry is on the link, job, step, or task library. The second parameter is the address in virtual storage of the list description field for the directory entry. The second two bytes at LISTADDR indicate the length of each entry. A character constant is

established to contain the directory information to be placed there by the control program as a result of the BLDL macro. The LINK macro in Figure 4-19 can now be written. Note that the DE parameter refers to the name field, not the list description field, of the directory entry.

```
LINK     DE=NAMEADDR,DCB=0,PARAM=(INDCB,OUTDCB,AREA),VL=1
```

Figure 4-19. The LINK Macro with a DE Parameter

Using CALL or Branch and Link

You can save time by passing control to a load module without using the control program. Pass control without using the control program as follows.

- Issue a LOAD macro to obtain a copy of the load module, preceded by a BLDL macro if you can shorten the search time by using it.

  The control program returns the address of the entry point and the addressing mode in register 0 and the length in doublewords in register 1.

- Load this address into register 15.

  The linkage requirements are the same when passing control between load modules as when passing control between control sections in the same load module: register 13 must contain a save area address, register 14 must contain the return address, and register 1 is used to pass parameters in a parameter list. A branch instruction, a branch and link instruction, a branch and save instruction, a branch and save and set mode instruction (BASSM), or a CALL macro can be used to pass control, using register 15. Use BASSM only if there is to be an addressing mode switch. The return will be made directly to your program.

Notes:

1. You must use a branch and save and set mode instruction if passing control to a module in a different addressing mode.
2. When control is passed to a load module without using the control program, you must check the load module attributes and current status of the copy yourself, and you must check the status in all succeeding uses of that load module during the job step, even when the control program is used to pass control.

The reason you have to keep track of the usability of the load module has been discussed previously; you are not allowing the control program to determine whether you can use a particular copy of the load module. The following paragraphs discuss your responsibilities when using load modules with various attributes. You must always know what the reusability attribute of the load module is. If you do not know, you should not attempt to pass control yourself.

If the load module is reenterable, one copy of the load module is all that is ever required for a job step. You do not have to determine the status of the copy; it can always be used. You can pass control by using a CALL macro, a branch, a branch and link instruction, a branch and save instruction, or a branch and save and set mode instruction (BASSM). Use BASSM only if there is to be an addressing mode switch.

If the load module is serially reusable, one use of the copy must be completed before the next use begins. If your job step consists of only one task, make sure that

the logic of your program does not require a second use of the same load module before completion of the first use. This prevents simultaneous use of the same copy. An exit routine must not require the use of a serially reusable load module also required in the main program.

Preventing simultaneous use of the same copy when you have more than one task in the job step requires more effort on your part. You must still be sure that the logic of the program for each task does not require a second use of the same load module before completion of the first use. You must also be sure that no more than one task requires the use of the same copy of the load module at one time. You can use the ENQ macro for this purpose. Properly used, the ENQ macro prevents the use of a serially reusable resource, in this case a load module, by more than one task at a time. For information on the ENQ macro, see Chapter 6, "Resource Control" on page 6-1. You can also use a conditional ENQ macro to check for simultaneous use of a serially reusable resource within one task.

If the load module is nonreusable, each copy can only be used once; you must be sure that you use a new copy each time you require the load module. You can ensure that you always get a new copy by using a LINK macro or by doing the following:

1. Issue a LOAD macro before you pass control.
2. Pass control using a branch, branch and link, branch and save, branch and save and set mode instruction, or a CALL macro.
3. Issue a DELETE macro as soon as you are through with the copy.

How Control is Returned

The return of control between load modules is the same as return of control between two control sections in the same load module. The program in the load module returning control is responsible for restoring registers 2-14, possibly loading a return code in register 15, passing control using the address in register 14 and possibly setting the correct addressing mode. The program in the load module to which control is returned can expect registers 2-13 to be unchanged, register 14 to contain the return address, and optionally, register 15 to contain a return code. Control can be returned using a branch instruction, a branch and set mode instruction or the RETURN macro. If control was passed without using the control program, control returns directly to the calling program. However, if control was originally passed using the control program, control returns first to the control program, then to the calling program.

The action taken by the control program is as follows. The control program returns in the caller's addressing mode. When control was passed using a LINK, LINKX, ATTACH, or ATTACHX macro, the responsibility count was increased by one for the copy of the load module to which control was passed to ensure that the copy would be in virtual storage as long as it was required. The return of control indicates to the control program that this use of the copy is completed, and so the responsibility count is decreased by one. The virtual storage area containing the copy is made available when the responsibility count reaches zero.

Passing Control without Return

Use the XCTL or XCTLX macro to pass control between load modules when no return of control is required. You can also pass control using a branch instruction. However, when you pass control in this manner, you must protect against multiple uses of nonreusable or serially reusable modules. The following paragraphs discuss the requirements for passing control without return in each case.

Passing Control Using a Branch Instruction

The same requirements and procedures for protecting against reuse of a nonreusable copy of a load module apply when passing control without return as were stated under "Passing Control With Return." The procedures for passing control are as follows.

Issue a LOAD macro to obtain a copy of the load module. The entry address and addressing mode returned in register 0 are loaded into register 15. The linkage requirements are the same when passing control between load modules as when passing control between control sections in the same load module; register 13 must be reloaded with the old save area address, then registers 14 and 2-12 restored from that old save area. Register 1 is used to pass parameters in a parameter list. If the addressing mode does not change, a branch instruction is issued to pass control to the address in register 15; if the addressing mode does change, a branch and save and set mode macro is used.

Note: Mixing branch instructions and XCTL or XCTLX macros is hazardous. The next topic explains why.

Using the XCTL or XCTLX Macro

The XCTL or XCTLX macro, in addition to being used to pass control, is used to indicate to the control program that this use of the load module containing the XCTL or XCTLX macro is completed. Because control is not to be returned, the address of the old save area must be reloaded into register 13. The return address must be loaded into register 14 from the old save area, as must the contents of registers 2-12. The XCTL or XCTLX macro can be written to request the loading of registers 2-12, or you can do it yourself. If you restore all registers yourself, do not use the EP parameter. This creates an inline parameter list that can only be addressed using your base register, and your base register is no longer valid. If EP is used, you must have XCTL or XCTLX restore the base register for you.

When using the XCTL or XCTLX macro, pass parameters in a parameter list. In this case, however, the parameter list (or the parameter data) must be established in a portion of virtual storage outside the current load module containing the XCTL or XCTLX macro. This is because the copy of the current load module may be deleted before the called load module can use the parameters, as explained in more detail below.

The XCTL or XCTLX macro is similar to the LINK macro in the method used to pass control: control is passed by way of the control program using a control program parameter list. The control program loads a copy of the load module, if necessary, loads the entry address in register 15, saves the address passed in register 14, and passes control to the address in register 15. The control program adds one to the responsibility count for the copy of the load module to which control is to be passed and subtracts one from the responsibility count for the current load module. The current load module in this case is the load module last given control using the control program in the performance of the active task. If you have been passing control between load modules without using the control program, chances are the responsibility count will be lowered for the wrong load module copy. And

remember, when the responsibility count of a copy reaches zero, that copy may be deleted, causing unpredictable results if you try to return control to it.

Figure 4-20 shows how this could happen. Control is given to load module A, which passes control to the load module B (step 1) using a LOAD macro and a branch and link instruction. Register 14 at this time contains the address of the instruction following the branch and link. Load module B then is executed, independently of how control was passed, and issues an XCTL or XCTLX macro when it is finished (step 2) to pass control to load module C. The control program knowing only of load module A, lowers the responsibility count of A by one, resulting in its deletion. Load module C is executed and returns to the address which used to follow the branch and link instruction. Step 3 of Figure 4-20 indicates the result.

Two methods are available for ensuring that the proper responsibility count is lowered. One way is to always use the control program to pass control with or without return. The other method is to use only LOAD and DELETE macros to determine whether or not a copy of a load module should remain in virtual storage.

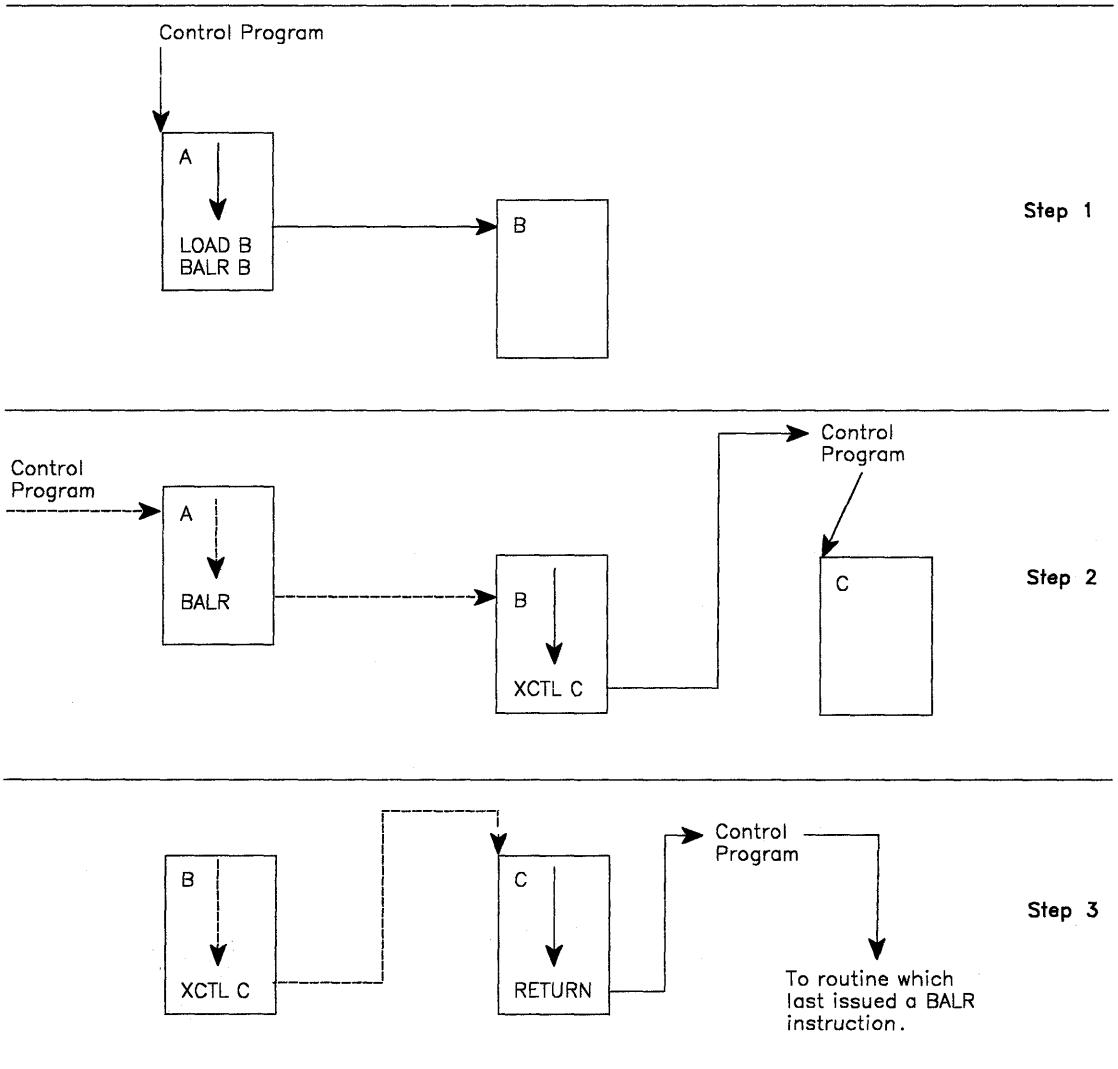


Figure 4-20. Misusing Control Program Facilities Causes Unpredictable Results

Additional Entry Points

Through the use of linkage editor facilities you can specify as many as 17 different names (a member name and 16 aliases) and associated entry points within a load module. It is only through the use of the member name or the aliases that a copy of the load module can be brought into virtual storage. Once a copy has been brought into virtual storage, however, additional entry points can be provided for the load module, subject to one restriction. The load module copy to which the entry point is to be added must be one of the following:

- A copy that satisfied the requirements of a LOAD macro issued during the same task
- The copy of the load module most recently given control through the control program in performance of the same task.

Add the entry point by using the IDENTIFY macro, which can be issued only by a program running under a program request block (PRB). The IDENTIFY macro cannot be issued by supervisor call routines or asynchronous exit routines established using other supervisor macros.

When you use the IDENTIFY macro, you specify the name to be used to identify the entry point, and the virtual storage address of the entry point in the copy of the load module. The address must be within a copy of a load module that meets the requirements listed above; if it is not, the entry point will not be added, and you will be given a return code of 0C (hexadecimal). The name can be any valid symbol of up to eight characters, and does not have to correspond to a name or symbol within the load module. The name must not be the same as any other name used to identify any load module available to the control program; duplicate names cause errors. The control program checks the names of all load modules in the link pack area, and the job pack area when you issue an IDENTIFY macro, and provides a return code of 8 if a duplicate is found. You are responsible for not duplicating a member name or an alias in any of the libraries.

IDENTIFY services sets the addressing mode of the alias entry point equal to the addressing mode of the major entry point.

If you create an alias for a module in the pageable link pack area, IDENTIFY services places an entry for the alias on the active link pack area queue.

Entry Point and Calling Sequence Identifiers as Debugging Aids

An entry point identifier is a character string of up to 70 characters that can be specified in a SAVE macro. The character string is created as part of the SAVE macro expansion.

A calling sequence identifier is a 16-bit binary number that can be specified in a CALL, LINK, or LINKX macro. When coded in a CALL, LINK, or LINKX macro, the calling sequence identifier is located in the two low-order bytes of the fullword at the return address. The high-order two bytes of the fullword form a NOP instruction.

Chapter 5. Understanding 31-Bit Addressing

Enterprise Systems Architecture*, like 370/Extended Architecture, supports 31-bit real and virtual addresses, which provide a maximum real and virtual address of two gigabytes (2^{31}) minus one. For compatibility with existing programs, MVS/ESA* and MVS/XA* also support 24-bit real and virtual addresses. The basic changes in the system that provide for both 31-bit addresses and the continued use of 24-bit addresses are:

- A virtual storage map of two gigabytes with MVS services to support programs executing or residing anywhere in virtual storage.
- Two program attributes that specify expected address length on entry and intended location in virtual storage.
- **Bimodal operation**, a capability of the processor that permits the execution of programs with 24-bit addresses as well as programs with 31-bit addresses.
- Instructions that are sensitive to addressing mode.

Virtual Storage

In the MVS virtual storage map:

- Each address space has its own two gigabytes of virtual storage.
- Each private area has a portion below 16 megabytes and an extended portion above 16 megabytes but, logically, these areas can be thought of as one area.

Figure 5-1 shows the virtual storage map.

Addressing Mode and Residency Mode

In MVS/ESA and MVS/XA, the processor can treat addresses as having either 24 or 31 bits. Addressing mode (AMODE) describes whether the processor is using 24-bit or 31-bit addresses. In MVS/ESA and MVS/XA, programs can reside in 24-bit addressable areas or beyond the 24-bit addressable area (above 16 megabytes). Residency mode (RMODE) specifies whether the program must reside in the 24-bit addressable area or can reside anywhere in 31-bit addressable storage.

Addressing mode (AMODE) and **residency mode** (RMODE) are program attributes specified (or defaulted) for each CSECT, load module, and load module alias. These attributes are the programmer's specification of the addressing mode in which the program is expected to get control and where the program is expected to reside in virtual storage.

AMODE defines the addressing mode (24, 31, or ANY) in which a program expects to receive control. Addressing mode refers to the address length that a program is prepared to handle on entry: 24-bit addresses, 31-bit addresses, or both (ANY). Programs with an addressing mode of ANY have been designed to receive control in either 24- or 31-bit addressing mode.

---

* Enterprise Systems Architecture is a trademark of the IBM Corporation.

* MVS/ESA is a trademark of the IBM Corporation.

* MVS/XA is a trademark of the IBM Corporation.

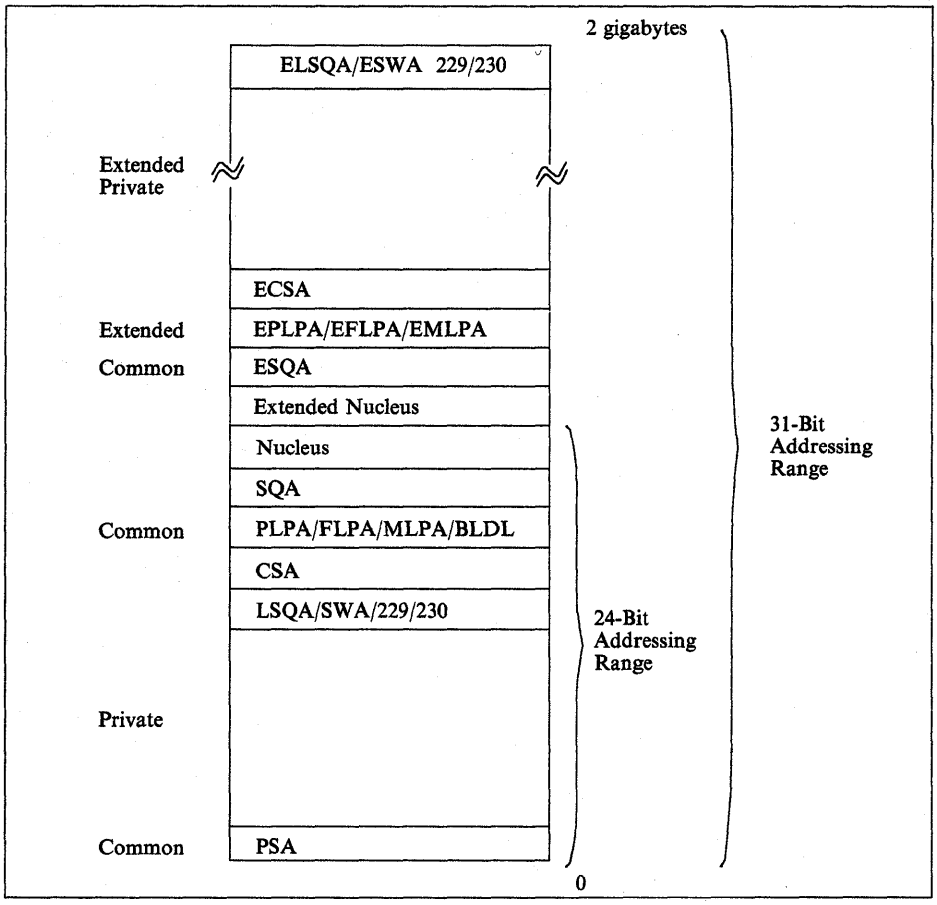


Figure 5-1. Two Gigabyte Virtual Storage Map

A 370-XA or 370-ESA processor can operate with either 24-bit addresses (16 megabytes of addressability) or 31-bit addresses (2 gigabytes of addressability). This ability of the processor to permit the execution of programs in 24-bit addressing mode as well as programs in 31-bit addressing mode is called **bimodal operation**. A program's AMODE attribute determines whether the program is to receive control with 24-bit or 31-bit addresses. Once a program gets control, the program can change the AMODE if necessary.

In 24-bit addressing mode, the processor treats all virtual addresses as 24-bit values. This makes it **impossible** for a program in 24-bit addressing mode to address virtual storage with an address greater than 16,777,215 (16 megabytes) because that is the largest number that a 24-bit binary field can contain.

In 31-bit addressing mode, the processor treats all virtual addresses as 31-bit values.

The processor supports bimodal operation so that both new programs and most old programs can execute correctly. Bimodal operation is necessary because certain coding practices in existing programs depend on 24-bit addresses. For example:

- Some programs use a 4-byte field for a 24-bit address and place flags in the high-order byte.
- Some programs use the LA instruction to clear the high-order byte of a register. (In 24-bit addressing mode, LA clears the high-order byte; in 31-bit addressing mode, it clears only the high-order bit.)
- Some programs depend on BAL and BALR to return the ILC (instruction length code), the CC (condition code), and the program mask. (BAL and BALR return this information in 24-bit addressing mode. In 31-bit addressing mode they do not.)

Each load module and each alias entry has an AMODE attribute.

A CSECT can have only one AMODE, which applies to all its entry points. Different CSECTs in a load module can have different AMODEs.

RMODE specifies where a program is expected to reside in virtual storage. The RMODE attribute is not related to central storage requirements. (RMODE 24 indicates that a program is coded to reside in virtual storage below 16 megabytes. RMODE ANY indicates that a program is coded to reside anywhere in virtual storage.)

Each load module and each alias entry has an RMODE attribute. The alias entry is assigned the same RMODE as the main entry.

The following kinds of programs must reside in the range of addresses below 16 megabytes (addressable by 24-bit callers):

- Programs that have the AMODE 24 attribute
- Programs that have the AMODE ANY attribute
- Programs that use system services that require their callers to be AMODE 24
- Programs that use system services that require their callers to be RMODE 24
- Programs that must be addressable by 24-bit addressing mode callers

Programs without these characteristics can reside anywhere in virtual storage.

"Addressing Mode and Residency Mode" on page 5-12
describes AMODE and RMODE processing and 31-bit addressing support of AMODE and RMODE in detail.

Requirements for Execution in 31-Bit Addressing Mode

In general, to execute in 31-bit addressing mode a program must:

- Be assembled using Assembler H Version 2 and the MVS/XA or MVS/ESA macro library.
- Be link edited using the linkage editor supplied with Data Facility Product (DFP) or be loaded using the loader supplied with DFP.
- Execute on an MVS/XA or MVS/ESA system.

Rules and Conventions for 31-Bit Addressing

It is important to distinguish the rules from the conventions when describing 31-bit addressing. There are only two rules, and they are associated with hardware:

1. The length of address fields is controlled by the A-mode bit (bit 32) in the PSW. When bit 32 = 1, addresses are treated as 31-bit values. When bit 32 = 0, addresses are treated as 24-bit values.

   Any data passed from a 31-bit addressing mode program to a 24-bit addressing mode program must reside in virtual storage below 16 megabytes. (A 24-bit addressing mode program cannot reference data above 16 megabytes without changing addressing mode.)

2. The A-mode bit affects the way some instructions work.

The conventions, on the other hand, are more extensive. Programs using system services must follow these conventions.

- A program must return control in the same addressing mode in which it received control.
- A program expects 24-bit addresses from 24-bit addressing mode programs and 31-bit addresses from 31-bit addressing mode programs.
- A program should validate the high-order byte of any address passed by a 24-bit addressing mode program before using it as an address in 31-bit addressing mode.

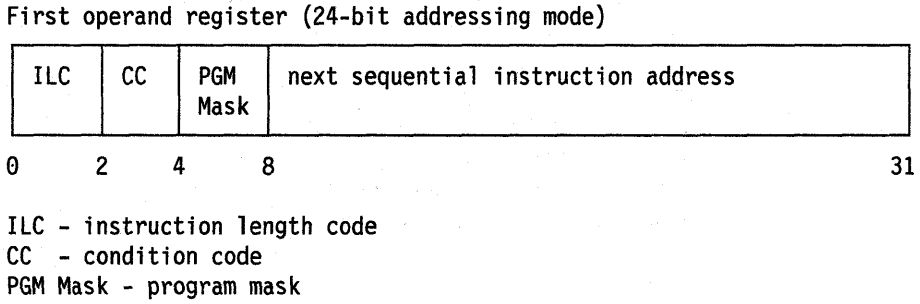
Mode Sensitive Instructions

The processor is sensitive to the addressing mode that is in effect (the setting of the PSW A-mode bit). The current PSW controls instruction sequencing. The instruction address field in the current PSW contains either a 24-bit address or a 31-bit address depending on the current setting of the PSW A-mode bit. For those instructions that develop or use addresses, the addressing mode in effect in the current PSW determines whether the addresses are 24 or 31 bits long.

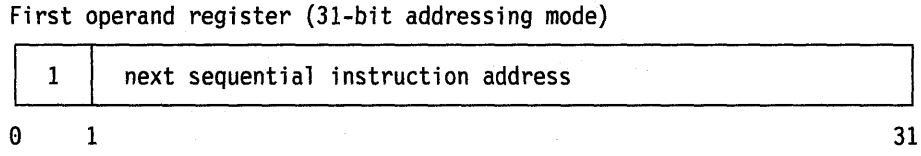
Principles of Operation contains a complete description of the 370-XA and 370-ESA instructions. The following topics provide an overview of the mode sensitive instructions.

BAL and BALR

BAL and BALR are addressing-mode sensitive. In 24-bit addressing mode, BAL and BALR work the same way as they do when executed on a processor running in 370 mode. BAL and BALR put link information into the high-order byte of the first operand register and put the return address into the remaining three bytes before branching.



In 31-bit addressing mode, BAL and BALR put the return address into bits 1 through 31 of the first operand register and save the current addressing mode in the high-order bit. Because the addressing mode is 31-bit, the high-order bit is always a 1.



When executing in 31-bit addressing mode, BAL and BALR do not save the instruction length code, the condition code, or the program mask. IPM* (insert program mask) can be used to save the program mask and the condition code.

LA: The LA (load address) instruction, when executed in 31-bit addressing mode, loads a 31-bit value and clears the high-order bit. When executed in 24-bit addressing mode, it loads a 24-bit value and clears the high-order byte (as in MVS/370 mode).

LRA: The LRA (load real address) instruction always results in a 31-bit real address regardless of the issuing program's AMODE. The virtual address specified is treated as a 24-bit or 31-bit address based on the value of the PSW A-mode bit at the time the LRA instruction is executed.

Branching Instructions

BASSM (branch and save and set mode) and BSM (branch and set mode) are branching instructions that manipulate the PSW A-mode bit (bit 32). Programs can use BASSM when branching to modules that might have different addressing modes. Programs invoked through a BASSM instruction can use a BSM instruction to return in the caller's addressing mode. BASSM and BSM are described in more detail in "Establishing Linkage" on page 5-24.

BAS (branch and save) and BASR:

- Save the return address and the current addressing mode in the first operand.
- Replace the PSW instruction address with the branch address.

The high-order bit of the return address indicates the addressing mode. BAS and BASR perform the same function that BAL and BALR perform in 31-bit addressing mode. In 24-bit mode, BAS and BASR put zeroes into the high-order byte of the return address register.

Use of 31-Bit Addressing

In addition to providing support for the use of 31-bit addresses by user programs, MVS includes many system services that use 31-bit addresses.

Some system services are independent of the addressing mode of their callers. These services accept callers in either 24-bit or 31-bit addressing mode and use 31-bit parameter address fields. They assume 24-bit addresses from 24-bit addressing mode callers and 31-bit addresses from 31-bit addressing mode callers. Most supervisor macros are in this category.

---

* IPM is a trademark of the IBM Corporation.

Other services have restrictions with respect to address parameter values. Some of these services accept SVC callers and allow them to be in either 24-bit or 31-bit addressing mode. However, the same services might require branch entry callers to be in 24-bit addressing mode or might require one or more parameter addresses to be less than 16 megabytes.

Some services do not support 31-bit addressing at all. To determine a service's addressing mode requirements, see the documentation that explains how to invoke the service. (VSAM accepts entry by a program that executes in either 24-bit or 31-bit addressing mode.) The *MVS Conversion Notebook for Version 2*, gives examples of the system services in each of these categories.

MVS provides instructions that support 31-bit addressing mode and bimodal operation. These instructions are supported only by Assembler H Version 2 installed with the ADV or UNIV instruction set specified. The linkage editor functions that support MVS are provided in Data Facility Product (DFP).

Planning for 31-Bit Addressing

Most programs that run on MVS/370 will run on MVS/XA or MVS/ESA in 24-bit addressing mode without change. Some programs need to be modified to execute in 31-bit addressing mode to provide the same function. Still other programs need to be modified to run in 24-bit addressing mode. The *MVS Conversion Notebook for Version 3* helps you identify programs that need to be changed. This section helps you determine what changes to make to a module you are converting to 31-bit addressing and indicates what 31-bit address-related things to consider when writing new code.

Some reasons for converting to 31-bit addressing mode are:

- The program can use more virtual storage for tables, arrays, or additional logic.
- The program needs to reference control blocks that have been moved above 16 megabytes.
- The program is invoked by other 31-bit addressing mode programs.
- The program must run in 31-bit addressing mode because it is a user exit routine that the system invokes in 31-bit mode.
- The program needs to invoke services that expect to get control in 31-bit addressing mode.

Converting Existing Programs

Keeping in mind that 31-bit addressing mode programs can reside either below or above 16 megabytes, you can convert existing programs as follows:

1. **Converting the program to use 31-bit addresses** - a change in addressing mode only.
   - You can change the entire module to use 31-bit addressing.
   - You can change only that portion that requires 31-bit addressing mode execution.

   Be sure to consider whether or not the code has any dependencies on 24-bit addresses. Such code does not produce the same results in 31-bit mode as it did in 24-bit mode. See "Mode Sensitive Instructions" on page 5-4 for an overview of instructions that function differently depending on addressing mode.

Figure 5-2 summarizes the things that you need to do to maintain the proper interface with a program that you plan to change to 31-bit addressing mode.

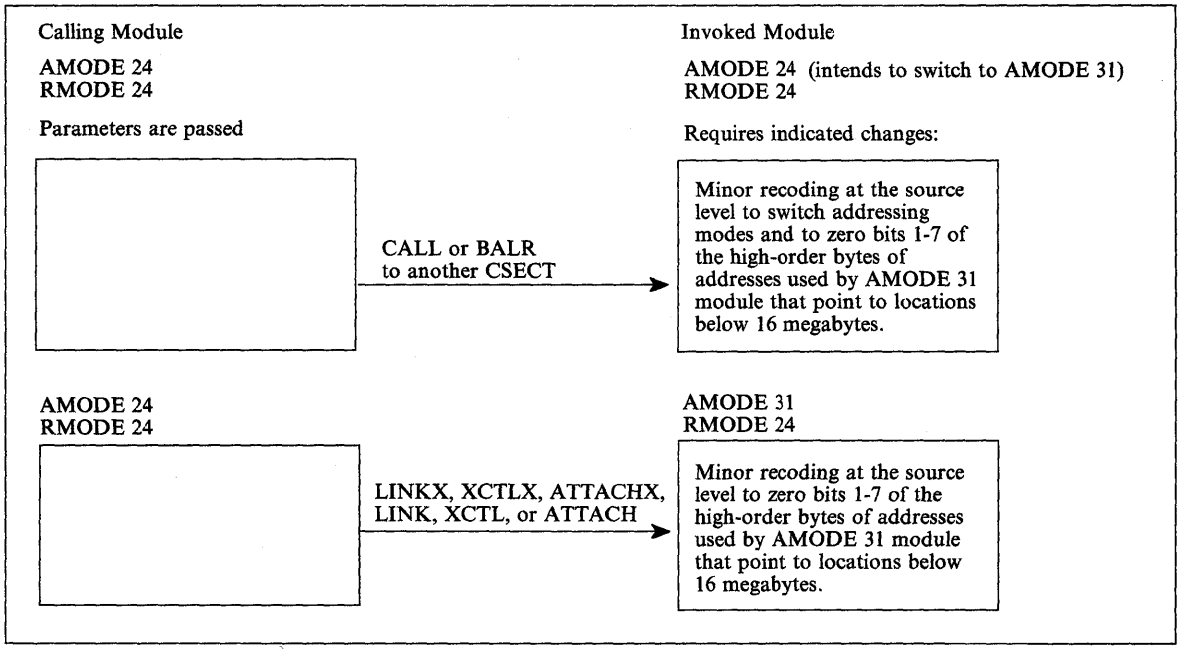


Figure 5-2. Maintaining Correct Interfaces to Modules that Change to AMODE 31

2. **Moving the program above 16 megabytes** - a change in both addressing mode and residency mode

In general, you move an existing program above 16 megabytes because there is not enough room for it below 16 megabytes. For example:

- An existing program or application is growing so large that soon it will not fit below 16 megabytes.
- An existing application that now runs as a series of separate programs, or that executes in an overlay structure, would be easier to manage as one large program.
- Code is in the system area, and moving it would provide more room for the private area below 16 megabytes.

The techniques used to establish proper interfaces to modules that move above 16 megabytes depend on the number of callers and the ways they invoke the module. Figure 5-3 summarizes the techniques for passing control. The programs involved must ensure that any addresses passed as parameters are treated correctly. (High-order bytes of addresses to be used by a 31-bit addressing mode program must be validated or zeroed.)

Means of Entry to Moved Module (AMODE 31,RMODE ANY)	Few AMODE 24,RMODE 24 Callers	Many AMODE 24,RMODE 24 Callers
LOAD macro and BALR	• Have caller use LINK OR LINKX or • Have caller use LOAD macro and BASSM (invoked program returns via BSM) or • Change caller to AMODE 31,RMODE 24 before BALR	Create a linkage assist routine (described in "Establishing Linkage" on page 5-24). Give the linkage assist routine the name of the moved module.
BALR using an address in a common control block	• Have caller switch to AMODE 31 when invoking or • Change the address in the control block to a pointer-defined value (described in "Establishing Linkage" on page 5-24) and use BASSM. (The moved module will use BSM to return.)	Create a linkage assist routine (described in "Establishing Linkage" on page 5-24).
ATTACH, ATTACHX, LINK, LINKX, XCTL, or XCTLX	No changes required.	No changes required.
SYNCH or SYNCHX in AMODE 24	• Have caller use SYNCH or SYNCHX with AMODE=31 parameter or • Have caller switch to AMODE 31 before issuing SYNCH or SYNCHX. • Change address in the control block to a pointer-defined value and use SYNCH or SYNCHX with AMODE=DEFINED.	Create a linkage assist routine (described in "Establishing Linkage" on page 5-24).

Figure 5-3. Establishing Correct Interfaces to Modules That Move Above 16 Megabytes

In deciding whether or not to modify a program to execute in 31-bit addressing mode either below or above 16 megabytes, there are several considerations:

1. How and by what is the module entered?
2. What system and user services does the module use that do not support 31-bit callers or parameters?
3. What kinds of coding practices does the module use that do not produce the same results in 31-bit mode as in 24-bit mode?
4. How are parameters passed? Can they reside above 16 megabytes?

Among the specific practices to check for are:

1. Does the module depend on the instruction length code, condition code, or program mask placed in the high order byte of the return address register by a 24-bit mode BAL or BALR instruction? One way to determine some of the dependencies is by checking all uses of the SPM (set program mask) instruction. SPM might indicate places where BAL or BALR were used to save the old program mask, which SPM might then have reset. The IPM (insert program mask) instruction can be used to save the condition code and the program mask.
2. Does the module use an LA instruction to clear the high-order byte of a register? This practice will not clear the high-order byte in 31-bit addressing mode.
3. Are any address fields that are less than 4 bytes still appropriate? Make sure that a load instruction does not pick up a 4-byte field containing a 3-byte address with extraneous data in the high-order byte. Make sure that bits 1-7 are zero.

4. Does the program use the ICM (insert characters under mask) instruction? The use of this instruction is sometimes a problem because it can put data into the high-order byte of a register containing an address, or it can put a 3-byte address into a register without first zeroing the register. If the register is then used as a base, index, or branch address register in 31-bit addressing mode, it might not indicate the proper address.
5. Does the program invoke 24-bit addressing mode programs? If so, shared data must be below 16 megabytes.
6. Is the program invoked by 24-bit or 31-bit addressing mode programs? Is the data in an area addressable by the programs that need to use it? (The data must be below 16 megabytes if used by a 24-bit addressing mode program.)

Writing New Programs That Use 31-Bit Addressing

You can write programs that execute in either 24-bit or 31-bit addressing mode. However, to maintain an interface with existing programs and with some system services, your 31-bit addressing mode programs need subroutines or portions of code that execute in 24-bit addressing mode. If your program resides below 16 megabytes, it can change to 24-bit addressing mode when necessary.

If your program resides above 16 megabytes, it needs a separate load module to perform the linkage to an unchanged 24-bit addressing mode program or service. Such load modules are called linkage assist routines and are described in "Establishing Linkage" on page 5-24.

When writing new programs, there are some things you can do to simplify the passing of parameters between programs that might be in different addressing modes. In addition, there are functions that you should consider and that you might need to accomplish your program's objectives. Following is a list of suggestions for coding programs to run on MVS/XA or MVS/ESA:

- Use fullword fields for addresses even if the addresses are only 24 bits in length.
- When obtaining addresses from 3-byte fields in existing areas, use SR (subtract register) to zero the register followed by ICM (insert characters under mask) in place of the load instruction to clear the high-order byte. For example:

```
Rather than:    L    1,A

        use:    SR   1,1
                ICM  1,7,A+1
```

  The 7 specifies a 4-bit mask of 0111. The ICM instruction shown inserts bytes beginning at location A+1 into register 1 under control of the mask. The bytes to be filled correspond to the 1 bits in the mask. Because the high-order byte in register 1 corresponds to the 0 bit in the mask, it is not filled.

- If the program needs storage above 16 megabytes, obtain the storage using the STORAGE macro (SP Version 3 only) or the VRU, VRC, RU, and RC forms of GETMAIN and FREEMAIN, or the corresponding functions on STORAGE. These are the only forms that allow you to obtain and free storage above 16 megabytes. Do not use storage areas above 16 megabytes for save areas and parameters passed to other programs.
- Do not use the STAE macro; use ESTAE or ESTAEX. STAE has 24-bit addressing mode dependencies.
- Do not use SPIE; use ESPIE. SPIE has 24-bit addressing mode dependencies.

- Do not use previous paging services macros; use PGSER.
- To make debugging easier, switch addressing modes only when necessary.
- Identify the intended AMODE and RMODE for the program in a prologue.
- 31-bit addressing mode programs should use ESTAE, ESTAEX or the ESTAI parameter on the ATTACH, or ATTACHX macro rather than STAE or STAI. STAI has 24-bit addressing mode dependencies. When recovery routines refer to the PSW field in the SDWA, they should refer to SDWAEC1, which is the EC mode PSW at the time of error.

  User-written STAE and STAI routines need to be aware of the restricted support of the BC mode PSW fields in the SDWA. The instruction length and address fields contain zeroes in the following situations:

  - SDWACTL1 (BC mode PSW at time of error) contains zeroes in the designated fields when the error occurred while the program (or a service routine executing on behalf of the program) was executing in 31-bit addressing mode.
  - SDWACTL2 (BC mode PSW from the last program request block (PRB) on the request block (RB) chain) contains zeroes when the last PRB on the RB chain refers to a program that was executing in 31-bit addressing mode.

When writing new programs, you need to decide whether to use 24-bit addressing mode or 31-bit addressing mode.

The following are examples of kinds of programs that you should write in 24-bit addressing mode:

- Programs that must execute on MVS/370 as well as MVS/XA or MVS/ESA and do not require any new MVS functions.
- Service routines, even those in the common area, that use system services requiring entry in 24-bit addressing mode or that must accept control directly from unchanged 24-bit addressing mode programs.

When you use 31-bit addressing mode, you must decide whether the new program should reside above or below 16 megabytes (unless it is so large that it will not fit below). Your decision depends on what programs and system services the new program invokes and what programs invoke it.

New Programs Below 16 Megabytes: The main reason for writing new 31-bit addressing mode programs that reside below 16 megabytes is to be able to address areas above 16 megabytes or to invoke 31-bit addressing mode programs while, at the same time, simplifying communication with existing 24-bit addressing mode programs or system services, particularly data management. For example, VSAM macros accept callers in 24-bit or 31-bit addressing mode.

Even though your program resides below 16 megabytes, you must be concerned about dealing with programs that require entry in 24-bit addressing mode or that require parameters to be below 16 megabytes. Figure 5-8 in "Establishing Linkage" on page 5-24 contains more information about parameter requirements.

New Programs Above 16 Megabytes: When you write new programs that reside above 16 megabytes, your main concerns are:

- Dealing with programs that require entry in 24-bit addressing mode or that require parameters to be below 16 megabytes. Note that these are concerns of any 31-bit addressing mode program no matter where it resides.
- How routines that remain below 16 megabytes invoke the new program.

Writing Programs for MVS/370 and MVS Systems with 31-Bit Addressing

You can write new programs that will run on both MVS/370 and MVS systems that use 31-bit addressing. If these programs do not need to use any new MVS functions, the best way to avoid errors is to assemble the programs on MVS/370 with macro libraries from a 31-bit addressing system (MVS/XA or MVS/ESA). You can also assemble these programs on 31-bit addressing systems with macro libraries from MVS/370, but you must generate MVS/370-compatible macro expansions by specifying the SPLEVEL macro at the beginning of the programs.

If the program needs to use new MVS functions, your programming task is more difficult because most new MVS/XA functions are not supported on MVS/370. You need to use dual paths in your program so that on each system your program uses the services or macros that are supported on that system.

Programs designed to execute on either 24 or 31-bit addressing systems must use fullword addresses where possible and use no new functions on macros except the LOC parameter on GETMAIN. These programs must also be aware of downward incompatible macros and use SPLEVEL as needed.

SPLEVEL Macro: Some macros are **downward incompatible**. The level of the macro expansion generated during assembly depends on the value of an assembler language global SET symbol. When the SET symbol value is 1, the system generates MVS/370 expansions. When the SET symbol value is 2 or greater, the system generates MVS/XA expansions.

The SPLEVEL macro allows programmers to change the value of the SET symbol. The SPLEVEL macro shipped with MVS/SP Version 3 sets a default value of 3 for the SET symbol. Therefore, unless a program or installation specifically changes the default value, the macros generated are MVS/ESA macro expansions.

You can, within a program, issue the SPLEVEL SET=1 macro to obtain MVS/370 (MVS/System Product Version 1 Release 3.0) expansions, or SPLEVEL SET=2 to obtain MVS/XA expansions. The SPLEVEL macro sets the SET symbol value for that program's assembly only and affects only the expansions within the program being assembled. A single program can include multiple SPLEVEL macros to generate different macro expansions. The following example shows how to obtain different macro expansions within the same program by assembling both expansions and making a test at execution time to determine which expansion to execute.

```
*    DETERMINE WHICH SYSTEM IS EXECUTING
          TM      CVTDCB,CVTMVSE (CVTMVSE is bit 0 in the
          BO      SP2            CVTDCB field. If bit 0=1,
                                 it indicates that MVS/XA
                                 is executing.)
*    INVOKE THE MVS/370 VERSION OF THE WTOR MACRO
          SPLEVEL SET=1
          WTOR    .....
          B       CONTINUE
SP2       EQU     *
*    INVOKE THE MVS/XA VERSION OF THE WTOR MACRO
          SPLEVEL SET=2
          WTOR    ......
CONTINUE  EQU     *
```

Authorized Assembler Programming Guide and *Authorized Assembler Programming Reference*, along with *Assembler Programming Guide* and *Assembler Programming Reference* describe the SPLEVEL macro.

Certain macros produce a "map" of control blocks or parameter lists. These mapping macros do not support the SPLEVEL macro. Mapping macros for different levels of MVS systems are available only in the macro libraries for each system. When programs use mapping macros, a different version of the program may be needed for each system.

Dual Programs: Sometimes two programs may be required, one for each system. In this case, use one of the following approaches:

- Keep each in a separate library
- Keep both in the same library but under different names

Addressing Mode and Residency Mode

Every program that executes in MVS/XA or MVS/ESA is assigned two program attributes: an addressing mode (AMODE) and a residency mode (RMODE). Programmers can specify these attributes for new programs. Programmers can also specify these attributes for old programs through reassembly, linkage editor PARM values, linkage editor MODE control statements, or loader PARM values. MVS assigns default attributes to any program that does not have AMODE and RMODE specified.

Addressing Mode - AMODE

AMODE is a program attribute that can be specified (or defaulted) for each CSECT, load module, and load module alias. AMODE states the addressing mode that is expected to be in effect when the program is entered. AMODE can have one of the following values:

- **AMODE 24** - The program is designed to receive control in 24-bit addressing mode.
- **AMODE 31** - The program is designed to receive control in 31-bit addressing mode.
- **AMODE ANY** - The program is designed to receive control in either 24-bit or 31-bit addressing mode.

Residency Mode - RMODE

RMODE is a program attribute that can be specified (or defaulted) for each CSECT, load module, and load module alias. RMODE states the virtual storage location (either above 16 megabytes or anywhere in virtual storage) where the program should reside. RMODE can have the following values:

- **RMODE 24** - The program is designed to reside below 16 megabytes in virtual storage. MVS places the program below 16 megabytes.
- **RMODE ANY** - The program is designed to reside at any virtual storage location, either above or below 16 megabytes. MVS places the program above 16 megabytes unless there is no suitable virtual storage above 16 megabytes.

AMODE and RMODE Combinations

Figure 5-4 shows all possible AMODE and RMODE combinations and indicates which are valid.

AMODE and RMODE Combinations at Execution Time

At execution time, there are only three valid AMODE/RMODE combinations:

1. AMODE 24, RMODE 24, which is the default
2. AMODE 31, RMODE 24
3. AMODE 31, RMODE ANY

The ATTACH, ATTACHX, LINK, LINKX, XCTL, and XCTLX macros give the invoked module control in the AMODE previously specified. However, specifying a particular AMODE does not guarantee that a module that gets control by other means will receive control in that AMODE. For example, an AMODE 24 module can issue a BALR to an AMODE 31, RMODE 24 module. The AMODE 31 module will get control in 24-bit addressing mode.

	RMODE 24	RMODE ANY
AMODE 24	Valid	Invalid 1
AMODE 31	Valid	Valid
AMODE ANY	Valid 2	It Depends 3

1. This combination is invalid because an AMODE 24 module cannot reside above 16 megabytes.
2. This is a valid combination in that the assembler, linkage editor, and loader accept it from all sources. However, the combination is not used at execution time. Specifying ANY is a way of deferring a decision about the actual AMODE until the last possible moment before execution. At execution time, however, the module must execute in either 24-bit or 31-bit addressing mode.
3. The attributes AMODE ANY/RMODE ANY take on a special meaning when used together. (This meaning might seem to disagree with the meaning of either taken alone.) A module with the AMODE ANY/RMODE ANY attributes will execute on either an MVS/370 or a system that uses 31-bit addressing (MVS/XA or MVS/ESA) if the module is designed to:
   - Use no facilities that are unique to MVS/XA or MVS/ESA.
   - Execute entirely in 31-bit addressing mode on a system that uses 31-bit addressing and return control to its caller in 31-bit addressing mode. (The AMODE could be different from invocation to invocation.)
   - Execute entirely in 24-bit addressing mode on an MVS/370 system.

   The linkage editor and loader accept this combination from the object module or load module but not from the PARM field of the linkage editor EXEC statement or the linkage editor MODE control statement. The linkage editor converts AMODE ANY/RMODE ANY to AMODE 31/RMODE ANY.

Figure 5-4. AMODE and RMODE Combinations

Determining the AMODE and RMODE of a Load Module

Use the AMBLIST service aid to find out the AMODE and RMODE of a load module. The module summary produced by the LISTLOAD control statement contains the AMODE of the main entry point and the AMODE of each alias, as well as the RMODE specified for the load module. Refer to *Service Aids* for information about AMBLIST.

You can look at the source code to determine the AMODE and RMODE that the programmer intended for the program. However, the linkage editor or the loader can override these specifications.

Assembler H Support of AMODE and RMODE

Assembler H Version 2 supports AMODE and RMODE assembler instructions. Using AMODE and RMODE assembler instructions, you can specify an AMODE and an RMODE to be associated with a control section, an unnamed control section, or a named common control section.

AMODE and RMODE in the Object Module

The assembler checks to determine if the specified AMODE/RMODE combination is valid. The only combination that is not valid is AMODE 24/ RMODE ANY.

The assembler also checks for the following error conditions:

- Multiple AMODE/RMODE statements for a single control section
- An AMODE/RMODE statement with an incorrect or missing value
- An AMODE/RMODE statement whose name field is not that of a valid control section in the assembly.

AMODE and RMODE Assembler Instructions

The AMODE instruction specifies the addressing mode to be associated with a CSECT in an object module. The format of the AMODE instruction is:

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

The name field associates the addressing mode with a control section. If there is a symbol in the name field of an AMODE statement, that symbol must also appear in the name field of a START, CSECT, or COM statement in the assembly. If the name field is blank, there must be an unnamed control section in the assembly.

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

Name	Operation	Operand
Any symbol or blank	RMODE	24/ANY

Both the RMODE and AMODE instructions can appear anywhere in the assembly. Their appearance does not initiate an unnamed CSECT. There can be more than one RMODE (or AMODE) instruction per assembly, but they must have different name fields.

The defaults when AMODE, RMODE, or both are not specified are:

Specified	**Defaulted**
Neither	AMODE 24 RMODE 24
AMODE 24	RMODE 24
AMODE 31	RMODE 24
AMODE ANY	RMODE 24
RMODE 24	AMODE 24
RMODE ANY	AMODE 31

DFP Linkage Editor Support of AMODE and RMODE

The linkage editor accepts AMODE and RMODE specifications from any or all of the following:

- Object modules.
- Load modules.
- PARM field of the linkage editor EXEC statement. For example:

  ```
  //LKED EXEC PGM=name,PARM='AMODE=31,RMODE=ANY,.....'
  ```

  PARM field input overrides object module and load module input.
- Linkage editor MODE control statements in the SYSLIN data set. For example:

  ```
  MODE AMODE(31),RMODE(24)
  ```

  MODE control statement input overrides object module, load module and PARM input.

Linkage editor processing results in two sets of AMODE and RMODE indicators located in:

- The load module
- The PDS entry for the member name and any PDS entries for alternate names or alternate entry points that were constructed using the linkage editor ALIAS control statement.

These two sets of indicators might differ because they can be created from different input. The linkage editor creates indicators in the load module based on input from the input object module and load module. The linkage editor creates indicators in the PDS directory based not only on input from the object module and load module but also on the PARM field of the linkage editor EXEC statement, and the MODE control statements in the SYSLIN data set. The last two sources of input override indicators from the object module and load module. Figure 5-5 shows linkage editor processing of AMODE and RMODE.

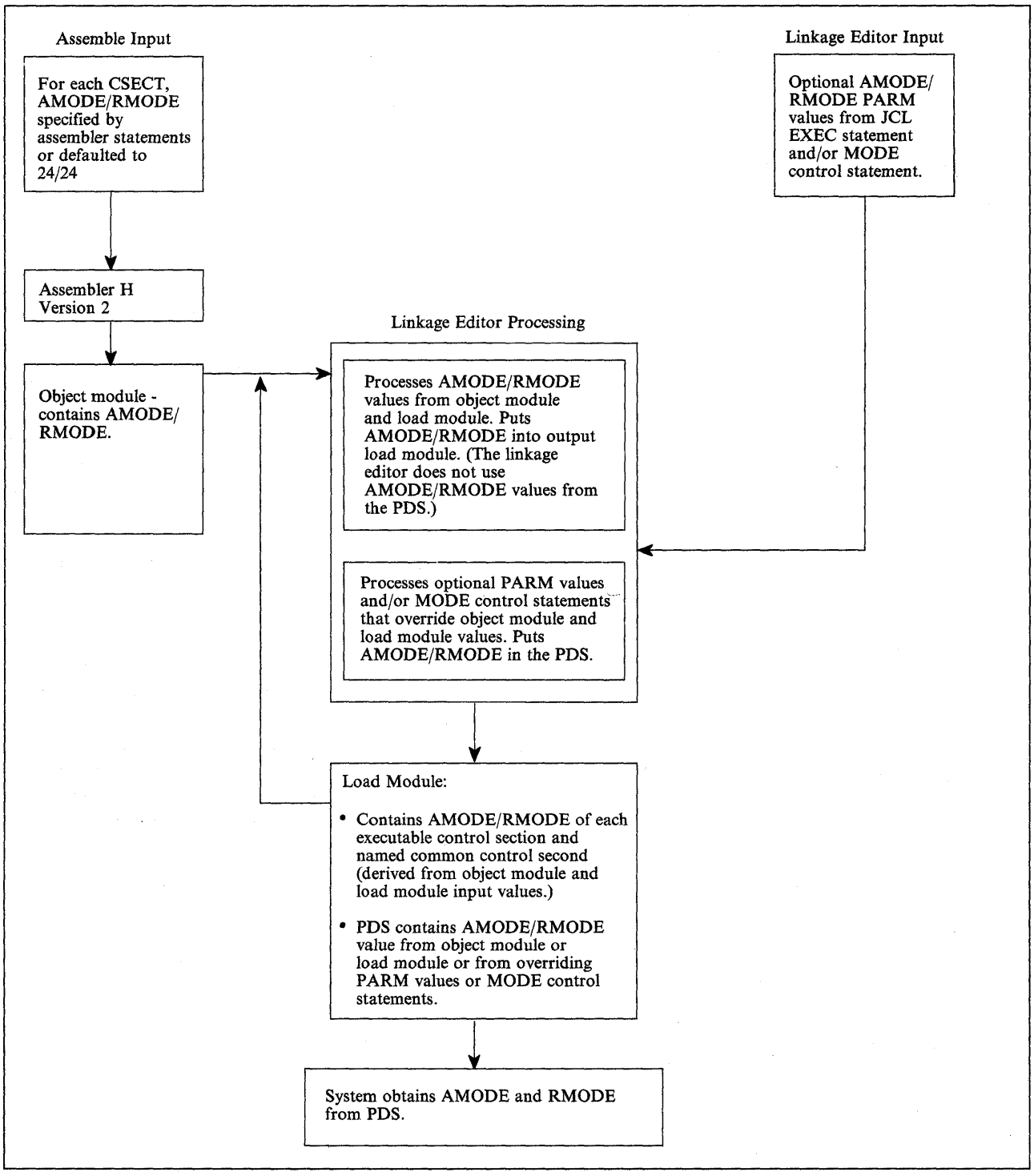


Figure 5-5. AMODE and RMODE Processing by the Linkage Editor

The linkage editor uses default values of AMODE 24/RMODE 24 for:

- Object modules produced by assemblers other than Assembler H Version 2
- Object modules produced by Assembler H Version 2 where source statements did not specify AMODE or RMODE
- Load modules produced by linkage editors other than the DFP linkage editor
- Load modules produced by the DFP linkage editor that did not have AMODE or RMODE specified from any input source
- Load modules in overlay structure.

MVS/XA and MVS/ESA treat programs in overlay structure as AMODE 24, RMODE 24 programs. Putting a program into overlay structure destroys any AMODE and RMODE specifications contained in the load module.

The linkage editor recognizes as valid the following combinations of AMODE and RMODE:

AMODE 24 RMODE 24

AMODE 31 RMODE 24

AMODE 31 RMODE ANY

AMODE ANY RMODE 24

AMODE ANY RMODE ANY

The linkage editor accepts the ANY/ANY combination from the object module or load module and places AMODE 31, RMODE ANY into the PDS (unless overridden by PARM values or MODE control statements). The linkage editor does not accept ANY/ANY from the PARM value or MODE control statement.

Any AMODE value specified alone in the PARM field or MODE control statement implies an RMODE of 24. Likewise, an RMODE of ANY specified alone implies an AMODE of 31. However, for RMODE 24 specified alone, the linkage editor does not assume an AMODE value. Instead, it uses the AMODE value specified in the CSECT in generating the entry or entries in the PDS.

When the linkage editor creates an overlay structure, it assigns AMODE 24, RMODE 24 to the resulting program.

Linkage Editor RMODE Processing

In constructing a load module, the linkage editor frequently is requested to combine multiple CSECTs, or it may process an existing load module as input, combining it with additional CSECTs or performing a CSECT replacement.

The linkage editor determines the RMODE of each CSECT. If the RMODEs are all the same, the linkage editor assigns that RMODE to the load module. If the RMODEs are not the same (ignoring the RMODE specification on common sections), the more restrictive value, RMODE 24, is chosen as the load module's RMODE.

The RMODE chosen can be overridden by the RMODE specified in the PARM field of the linkage editor EXEC statement. Likewise, the PARM field RMODE can be overridden by the RMODE value specified on the linkage editor MODE control statement.

The linkage editor does not alter the RMODE values obtained from the object module or load module when constructing the new load module. Any choice that the linkage editor makes or any override processing that it performs affects only the PDS.

DFP Loader Support for AMODE and RMODE

The loader's processing of AMODE and RMODE is similar to the linkage editor's. The loader accepts AMODE and RMODE specifications from:

Object modules
Load modules
PARM field of the JCL EXEC statement

Unlike the linkage editor, the loader does not accept MODE control statements from the SYSLIN data set, but it does base its loading sequence on the sequence of items in SYSLIN.

The loader passes the AMODE value to MVS. The loader processes the RMODE value as follows. If the user specifies an RMODE value in the PARM field, that value overrides any previous RMODE value. Using the value of the first RMODE it finds in the first object module or load module it encounters that is not for a common section, the loader obtains virtual storage for its output. As the loading process continues, the loader may encounter a more restrictive RMODE value. If, for example, the loader begins loading based on an RMODE ANY indicator and later finds an RMODE 24 indicator in a section other than a common section, it issues a message and starts over based on the more restrictive RMODE value. Figure 5-6 shows loader processing of AMODE and RMODE.

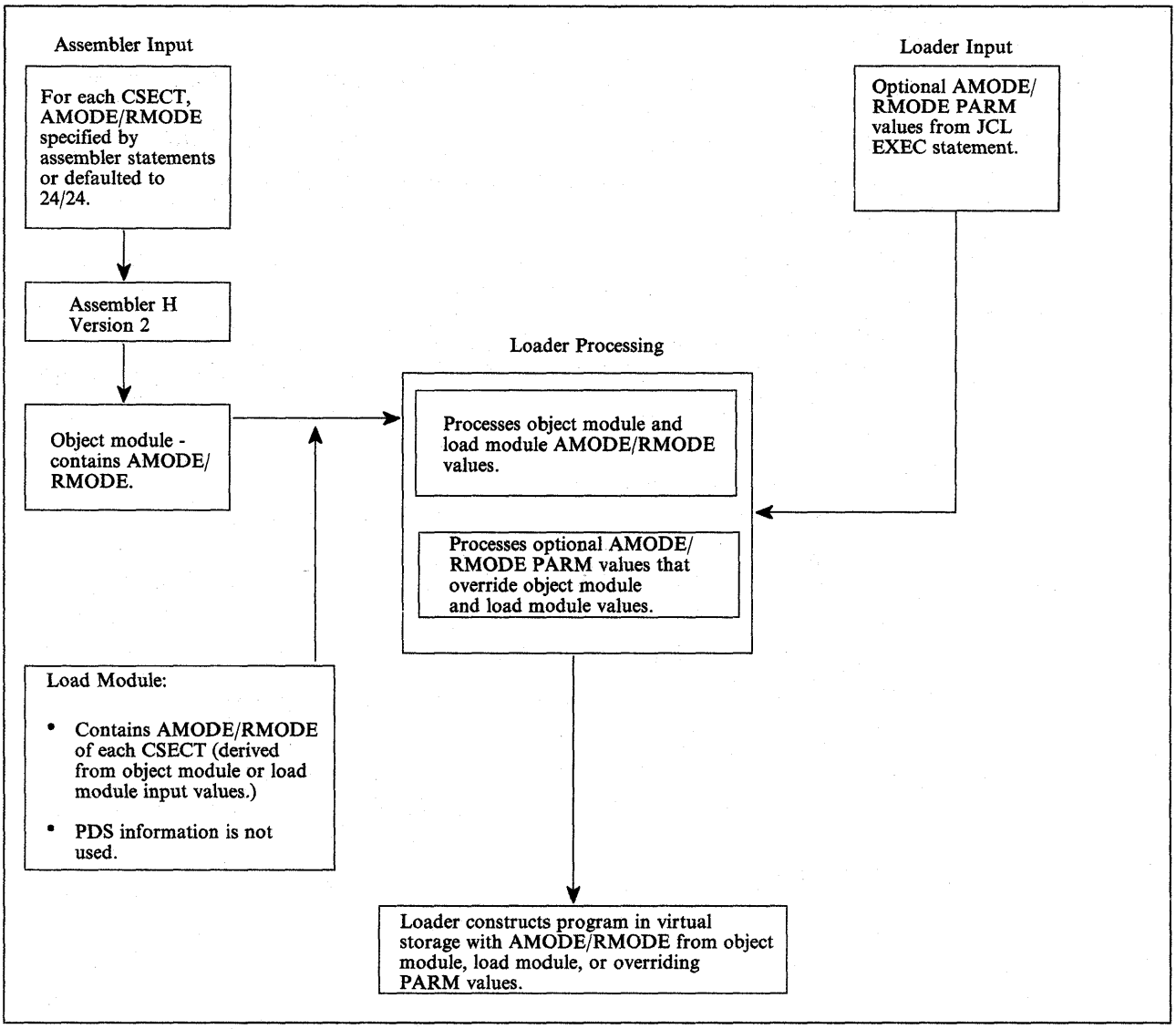


Figure 5-6. AMODE and RMODE Processing by the Loader

MVS Support of AMODE and RMODE

The following are examples of MVS support of AMODE and RMODE:

- MVS obtains storage for the module as indicated by RMODE.
- ATTACH, ATTACHX, LINK, LINKX, XCTL, and XCTLX give the invoked module control in the addressing mode specified by its AMODE.
- LOAD brings a module into storage based on its RMODE and sets bit 0 in register 0 to indicate its AMODE.
- CALL passes control in the AMODE of the caller.
- SYNCH or SYNCHX has an AMODE parameter that you can use to specify the AMODE of the invoked module.
- For SVC's, the system saves and sets the addressing mode.
- SRBs are dispatched in the addressing mode indicated by the SRB specified to the SCHEDULE macro.

- The cross memory instructions PC and PT establish the addressing mode for the target program.
- DFP access methods, except VSAM macros and OPEN and CLOSE macros, support AMODE 24 RMODE 24 callers only. VSAM macros and OPEN and CLOSE macros support all addressing and residency mode callers.
- Dumping is based on the AMODE specified in the error-related PSW.

Program Fetch

The system uses RMODE information from the PDS to determine whether to obtain storage above or below 16 megabytes.

ATTACH, ATTACHX, LINK, LINKX, XCTL, and XCTLX

Issuing an ATTACH or ATTACHX macro causes the control program to create a new task and indicates the entry point to be given control when the new task becomes active. If the entry point is a member name or an alias in the PDS. ATTACH or ATTACHX gives it control in the addressing mode specified in the PDS or in the mode specified by the loader. If the invoked program has the AMODE ANY attribute, it gets control in the AMODE of its caller.

The LINK, LINKX, XCTL, and XCTLX macros also give the invoked program control in the addressing mode indicated by its PDS for programs brought in by fetch or in the AMODE specified by the loader. The entry point specified must be a member name or an alias in the PDS passed by the loader, or specified in an IDENTIFY macro. If the entry point is an entry name specified in an IDENTIFY macro, IDENTIFY sets the addressing mode of the entry name equal to the addressing mode of the main entry point.

LOAD

Issuing the LOAD macro causes MVS to bring the load module containing the specified entry point name into virtual storage (if a usable copy is not already there). LOAD sets the high-order bit of the entry point address in register 0 to indicate the module's AMODE (0 for 24, 1 for 31), which LOAD obtains from the module's PDS entry. If the module's AMODE is ANY, LOAD sets the high-order bit in register 0 to correspond to the caller's AMODE.

LOAD places the module in virtual storage either above or below 16 megabytes as indicated by the module's RMODE, which is specified in the PDS for the module.

Specifying the ADDR parameter indicates that you want the module loaded at a particular location. If you specify an address above 16 megabytes, be sure that the module being loaded has the RMODE ANY attribute. If you do not know the AMODE and RMODE attributes of the module, specify an address below 16 megabytes or omit the ADDR parameter.

CALL

The CALL macro passes control to an entry point via BALR. Thus control is transferred in the AMODE of the caller. CALL does not change AMODE.

SYNCH or SYNCHX

Using the AMODE parameter on the SYNCH or SYNCHX macro, you can specify the addressing mode in which the invoked module is to get control. Otherwise, SYNCH or SYNCHX passes control in the caller's addressing mode.

SVC

For SVCs (supervisor calls), MVS saves and restores the issuer's addressing mode and makes sure that the invoked service gets control in the specified addressing mode.

SRB

When an SRB (service request block) is dispatched, MVS sets the addressing mode based on the high-order bit of the SRBEP field. This bit, set by the issuer of the SCHEDULE macro, indicates the addressing mode of the routine operating under the dispatched SRB.

PC and PT

For a program call (PC), the entry table indicates the target program's addressing mode. The address field in the entry table must be initialized by setting the high-order bit to 0 for 24-bit addressing mode or to 1 for 31-bit addressing mode.

The PC instruction sets up register 14 with the return address and AMODE for use with the PT (program transfer) instruction. If PT is not preceded by a PC instruction, the PT issuer must set the high-order bit of the second operand register to indicate the AMODE of the program being entered (0 for 24-bit addressing mode or 1 for 31-bit addressing mode).

Data Management Access Methods

User programs must be in AMODE 24, RMODE 24 when invoking DFP access methods other than VSAM. All non-VSAM access methods require parameter lists, control blocks, buffers, and user exit routines to reside in virtual storage below 16 megabytes.

VSAM request macros accept callers in AMODE 31, RMODE ANY. VSAM allows parameter lists and control blocks to reside above 16 megabytes; for details on addressing and residence requirements for VSAM parameter lists, control blocks, buffers, and exit routines, see *Managing VSAM Data Sets*.

AMODE's Effect on Dumps

The only time AMODE has an effect on dumps is when data on either side of the address in each register is dumped. If the addresses in registers are treated as 24-bit addresses, the data dumped may come from a different storage location than when the addresses are treated as 31-bit addresses. If a dump occurs shortly after an addressing mode switch, some registers may contain 31-bit addresses and some may contain 24 bit addresses, but dumping services does not distinguish among them. Dumping services uses the AMODE from the error-related PSW. For example, in dumping the area related to the registers saved in the SDWA, dumping services uses the AMODE from the error PSW stored in the SDWA.

How to Change Addressing Mode

To change addressing mode you must change the value of the PSW A-mode bit. The following list includes all the ways to change addressing mode.

- The mode setting instructions BASSM and BSM.
- Macros (ATTACH, ATTACHX, LINK, LINKX, XCTL, or XCTLX). The system makes sure that routines get control in the specified addressing mode. Users need only ensure that parameter requirements are met. MVS restores the invoker's mode on return from LINK or LINKX.
- SVCs. The supervisor saves and restores the issuer's addressing mode and ensures that the service routine receives control in the addressing mode specified in its SVC table entry.
- SYNCH or SYNCHX with the AMODE parameter to specify the addressing mode in which the invoked routine is to get control.

- An SRB. When the SRB is dispatched, the system sets the PSW A-mode bit with the high-order bit of the SRBEP field.
- The CIRB macro and the stage 2 exit effector. The CIRB macro is described in *Authorized Assembler Programming Guide* and *Authorized Assembler Programming Reference.*
- A PC, PT, or PR instruction. These three instructions establish the specified addressing mode.
- An LPSW instruction (not recommended).

The example in Figure 5-7 illustrates how a change in addressing mode in a 24-bit addressing mode program enables the program to retrieve data from the ACTLB control block, which might reside above 16 megabytes. The example works correctly whether or not the control block is actually above 16 megabytes. The example uses the BSM instruction to change addressing mode. In the example, the instruction L 2,4(,15) must be executed in 31-bit addressing mode. Mode setting code (BSM) before the instruction establishes 31-bit addressing mode and code following the instruction establishes 24-bit addressing mode.

```
USER     CSECT
USER     RMODE 24
USER     AMODE 24
         L   15,ACTLB
         L   1,LABEL1    SET HIGH-ORDER BIT OF REGISTER 1 TO 1
                         AND PUT ADDRESS INTO BITS 1-31
         BSM 0,1         SET AMODE 31 (DOES NOT PRESERVE AMODE)
LABEL1   DC  A(LABEL2 + X'80000000')
LABEL2   DS  0H
         L   2,4(,15)    OBTAIN DATA FROM ABOVE 16 MEGABYTES
         LA  1,LABEL3    SET HIGH-ORDER BIT OF REGISTER 1 TO 0
                         AND PUT ADDRESS INTO BITS 1-31
         BSM 0,1         SET AMODE 24 (DOES NOT PRESERVE AMODE)
LABEL3   DS  0H
```

Figure 5-7. Mode Switching to Retrieve Data from Above 16 Megabytes

Establishing Linkage

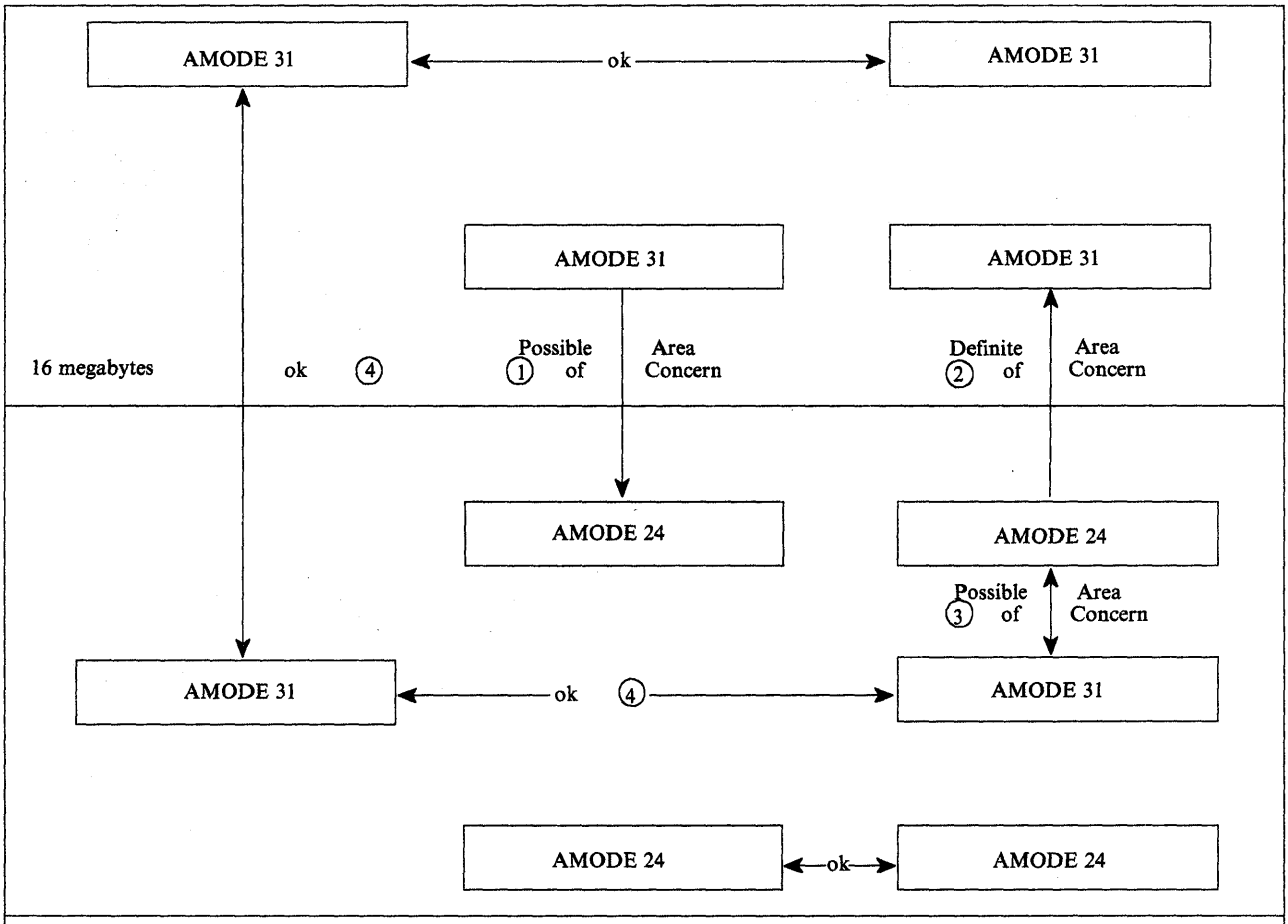
This section describes the mechanics of correct linkage in 31-bit addressing mode. Keep in mind that there are considerations other than linkage, such as locations of areas that both the calling module and the invoked module need to address.

Linkage in MVS systems that use 31-bit addressing (MVS/XA and MVS/ESA) is the same as in MVS/370 for modules whose addressing modes are the same. As shown in Figure 5-8, it is the linkage between modules whose addressing modes are different that is an area of concern. The areas of concern that appear in Figure 5-8 fall into two basic categories:

- Addresses passed as parameters from one routine to another must be addresses that both routines can use.
  - High-order bytes of addresses must contain zeroes or data that the receiving routine is programmed to expect.
  - Addresses must be less than 16 megabytes if they could be passed to a 24-bit addressing mode program.
- On transfers of control between programs with different AMODEs, the receiving routine must get control in the AMODE it needs and return control to the calling routine in the AMODE the calling routine needs.

There are a number of ways of dealing with the areas of concern that appear in Figure 5-8:

- Use the branching instructions (BASSM and BSM)
- Use pointer-defined linkage
- Use supervisor-assisted linkage (ATTACH, ATTACHX, LINK, LINKX, XCTL, and XCTLX)
- Use linkage assist routines
- Use "capping."



1. When an AMODE 31 module that resides above 16 megabytes invokes an AMODE 24 module, the concerns are:
   - The AMODE 24 program needs to receive control 24-bit mode.
   - The location of shared data (including control blocks, register save areas, and parameters). Can the AMODE 24 module address the data?
   - The AMODE 24 module cannot return control unless an addressing mode change occurs.
2. An AMODE 24 module cannot invoke an AMODE 31 module that resides above the line unless the AMODE 24 module changes its addressing mode either directly or using supervisor-assisted linkage.
3. When both modules are below 16 megabytes the concerns are:
   - Which module cleans out bits 1-7 of the high-order bytes of 24-bit values used as addresses?
   - Can both modules address shared data?
4. While there are no restrictions on the mechanics of linkage between two AMODE 31 modules, there might be restrictions on parameter values.

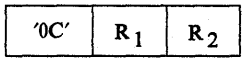
Figure 5-8. Linkage Between Modules with Different AMODEs and RMODEs

Using the BASSM and BSM Instructions

The BASSM (branch and save and set mode) and the BSM (branch and set mode) instructions are branching instructions that set the addressing mode. They are designed to complement each other. (BASSM is used to call and BSM is used to return, but they are not limited to such use.)

The description of BASSM appears in Figure 5-9. (See *Principles of Operation* for more information.)

BASSM R_1,R_2 [RR]



Bits 32-63 of the current PSW, including the updated instruction address, are saved as link information in the general register designated by R_1. Subsequently, the addressing mode and instruction address in the current PSW are replaced from the second operand. The action associated with the second operand is not performed if the R_2 field is zero.

The contents of the general register designated by the R_2 field specify the new addressing mode and branch address; however when the R_2 field is zero, the operation is performed without branching and without setting the addressing mode.

When the contents of the general register designated by the R_2 field are used, bit 0 of the register specifies the new addressing mode and replaces bit 32 of the current PSW, and the branch address is generated from the contents of the register under the control of the new addressing mode. The new value for the PSW is computed before the register designated by R_1 is changed.

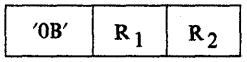
Condition Code: The code remains unchanged.

Program Exceptions: Trace (R_2 field is not zero).

Figure 5-9. BRANCH and SAVE and Set Mode Description

The description of BSM appears in Figure 5-10. (See *Principles of Operation* for more information.)

BSM R_1,R_2 [RR]



Bit 32 of the current PSW, the addressing mode, is inserted into the first operand. Subsequently the addressing mode and instruction address in the current PSW are replaced from the second operand. The action associated with an operand is not performed if the associated R field is zero.

The value of bit 32 of the PSW is placed in bit position 0 of the general register designated by R_1, and bits 1-31 of the register remain unchanged; however, when the R_1 field is zero, the bit is not inserted, and the contents of general register 0 are not changed.

The contents of the general register designated by the R_2 field specify the new addressing mode and branch address; however, when the R_2 field is zero, the operation is performed without branching and without setting the addressing mode.

When the contents of the general register designated by the R_2 field are used, bit 0 of the register specifies the new addressing mode and replaces bit 32 of the current PSW, and the branch address is generated from the contents of the register under the control of the new addressing mode. The new value for the PSW is computed before the register designated by R_1 is changed.

Condition Code: The code remains unchanged.

Program Exceptions: None.

Figure 5-10. Branch and Set Mode Description

Calling and Returning with BASSM and BSM

In the following example, a module named BELOW has the attributes AMODE 24, RMODE 24. BELOW uses a LOAD macro to obtain the address of module ABOVE. The LOAD macro returns the address in register 0 with the addressing mode indicated in bit 0 (a pointer-defined value). BELOW stores this address in location EPABOVE. When BELOW is ready to branch to ABOVE, BELOW loads ABOVE's entry point address from EPABOVE into register 15 and branches using BASSM 14,15. BASSM places the address of the next instruction into register 14 and sets bit 0 in register 14 to 0 to correspond to BELOW's addressing mode. BASSM replaces the PSW A-mode bit with bit 0 of register 15 (a 1 in this example) and replaces the PSW instruction address with the branch address (bits 1-31 of register 15) causing the branch.

ABOVE uses a BSM 0,14 to return. BSM 0,14 does not save ABOVE's addressing mode because 0 is specified as the first operand register. It replaces the PSW A-mode bit with bit 0 of register 14 (BELOW's addressing mode set by BASSM) and branches.

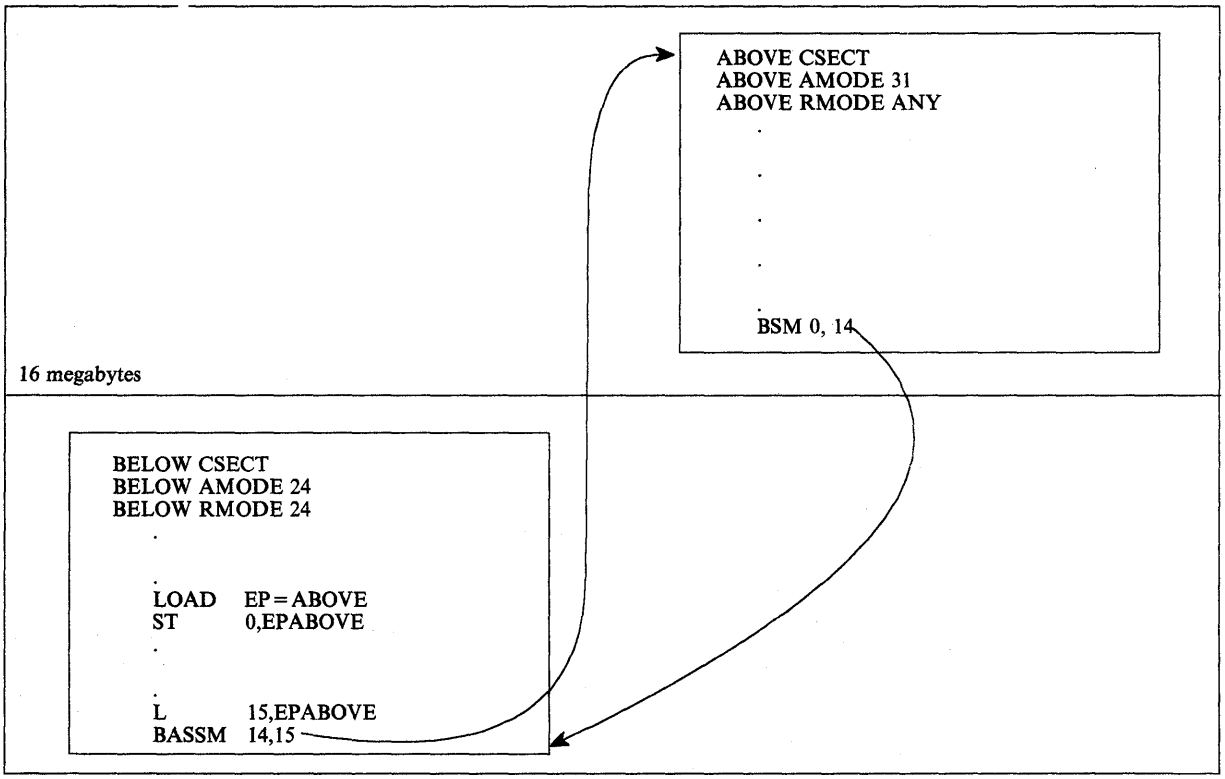


Figure 5-11. Using BASSM and BSM

Using Pointer-Defined Linkage

Pointer-defined linkage is a convention whereby programs can transfer control back and forth without having to know each other's AMODEs. Pointer-defined linkage is simple and efficient. You should use it in new or modified modules where there might be mode switching between modules.

Pointer-defined linkage uses a **pointer-defined value**, which is a 4-byte area that contains both an AMODE indicator and an address. The high-order bit contains the AMODE; the remainder of the word contains the address. To use pointer-defined linkage, you must:

- Use a pointer-defined value to indicate the entry point address and the entry point's AMODE. (The LOAD macro provides a pointer-defined value.)
- Use the BASSM instruction specifying a register that contains the pointer-defined value. BASSM saves the caller's AMODE and next the address of the next sequential instruction, sets the AMODE of the target routine, and branches to the specified location.
- Have the target routine save the full contents of the return register and use it in the BSM instruction to return to the caller.

Using an ADCON to Obtain a Pointer-Defined Value: The following method is useful when you need to construct pointer-defined values to use in pointer-defined linkages between control sections or modules that will be link edited into a single load module. You can also use this method when the executable program is prepared in storage using the loader.

The method requires the use of an externally-defined address constant in the routine to be invoked that identifies its entry mode and address. The address constant must contain a pointer-defined value. The calling program loads the pointer-defined value and uses it in a BASSM instruction. The invoked routine returns using a BSM instruction.

In Figure 5-12, RTN1 obtains pointer-defined values from RTN2 and RTN3. RTN1, the invoking routine does not have to know the addressing modes of RTN2 and RTN3. Later, RTN2 or RTN3 could be changed to use different addressing modes, and at that time their address constants would be changed to correspond to their new addressing mode. RTN1, however, would not have to change the sequence of code it uses to invoke RTN2 and RTN3.

You can use the techniques that the previous example illustrates to handle routines that have multiple entry points (possibly with different AMODE attributes). You need to construct a table of address constants, one for each entry point to be handled.

```
RTN1    CSECT
        EXTRN   RTN2AD
        EXTRN   RTN3AD
        .
        .
        L       15,=A(RTN2AD)   LOAD ADDRESS OF POINTER-DEFINED VALUE
        L       15,0(,15)       LOAD POINTER-DEFINED VALUE
        BASSM   14,15           GO TO RTN2 VIA BASSM
        .
        .
        L       15,=A(RTN3AD)   LOAD ADDRESS OF POINTER-DEFINED VALUE
        L       15,0(,15)       LOAD POINTER DEFINED-VALUE
        BASSM   14,15           GO TO RTN3 VIA BASSM
        .
```

```
RTN2    CSECT
RTN2    AMODE   24
        ENTRY   RTN2AD
        .
        BSM     0,14            RETURN TO CALLER IN CALLER'S MODE
RTN2AD  DC      A(RTN2)         WHEN USED AS A POINTER-DEFINED VALUE,
                                INDICATES AMODE 24 BECAUSE BIT 0 IS 0
```

```
RTN3    CSECT
RTN3    AMODE   31
        ENTRY   RTN3AD
        .
        BSM     0,14                RETURN TO CALLER IN CALLER'S MODE
RTN3AD  DC      A(X'80000000'+RTN3) WHEN USED AS A POINTER-DEFINED VALUE
                                    INDICATES AMODE 31 BECAUSE BIT 0 IS 1
```

Figure 5-12. Example of Pointer-Defined Linkage

As with all forms of linkage, there are considerations over and above the linkage mechanism. These include:

- Both routines must have addressability to any parameters passed.
- Both routines must agree which of them will clean up any 24-bit addresses that might have extraneous information bits 1-7 of the high-order byte. (This is a consideration only for AMODE 31 programs.)

When a 24-bit addressing mode program invokes a module that is to execute in 31-bit addressing mode, the calling program must ensure that register 13 contains a valid 31-bit address of the register save area with no extraneous data in bits 1-7 of the high-order byte. In addition, when any program invokes a 24-bit addressing mode program, register 13 must point to a register save area located below 16 megabytes.

Using the LOAD Macro to Obtain a Pointer-Defined Value: LOAD returns a pointer-defined value in register 0. You can preserve this pointer-defined value and use it with a BASSM instruction to pass control without having to know the target routine's AMODE.

Using Supervisor-Assisted Linkage

Figure 5-13 shows a "before" and "after" situation involving two modules, MOD1 and MOD2. In the BEFORE part of the figure both modules execute in 24-bit addressing mode. MOD1 invokes MOD2 using the LINK or LINKX macro. The AFTER part of the figure shows MOD2 moving above 16 megabytes and outlines the steps that were necessary to make sure both modules continue to perform their previous function.

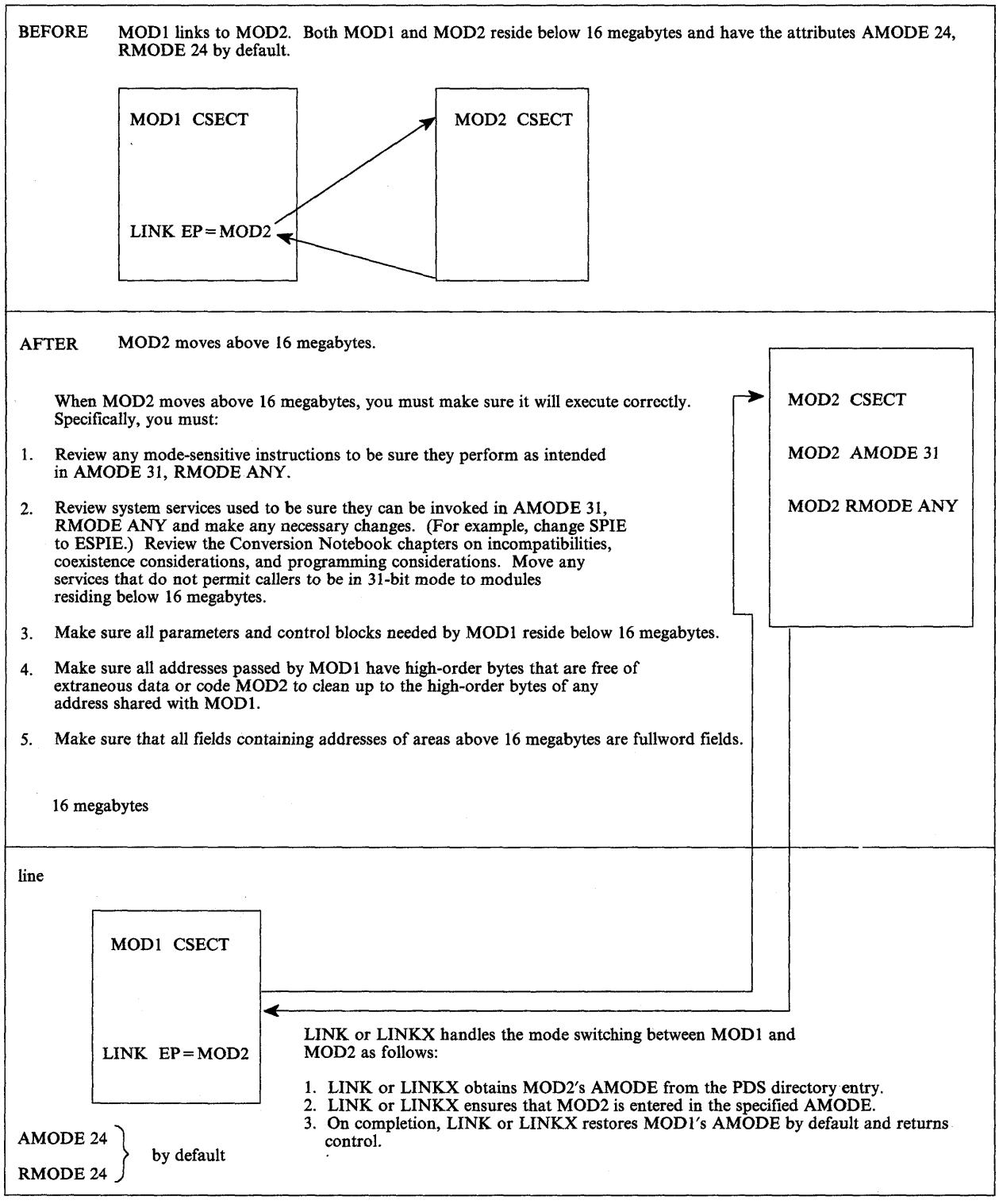


Figure 5-13. Example of Supervisor-Assisted Linkage

Linkage Assist Routines

A **linkage assist routine**, sometimes called an addressing mode interface routine, is a module that performs linkage for modules executing in different addressing or residency modes. Using a linkage assist routine, a 24-bit addressing mode module can invoke a 31-bit addressing mode module without having to make any changes. The invocation results in an entry to a linkage assist routine that resides below 16 megabytes and invokes the 31-bit addressing mode module in the specified addressing mode.

Conversely, a 31-bit addressing mode module, such as a new user module, can use a linkage assist routine to communicate with other user modules that execute in 24-bit addressing mode. The caller appears to be making a direct branch to the target module, but branches instead to a linkage assist routine that changes modes and performs the branch to the target routine.

The main advantage of using a linkage assist routine is to insulate a module from addressing mode changes that are occurring around it.

The main disadvantage of using a linkage assist routine is that it adds overhead to the interface. In addition, it takes time to develop and test the linkage assist routine. Some alternatives to using linkage assist routines are:

- Changing the modules to use pointer-defined linkage (described in "Using Pointer-Defined Linkage" on page 5-28).
- Adding a prologue and epilogue to a module to handle entry and exit mode switching, as described later in this chapter under "Capping."

Example of Using a Linkage Assist Routine

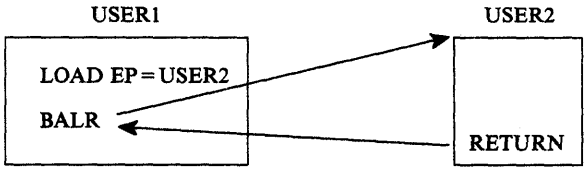
Figure 5-14 shows a "before" and "after" situation involving modules USER1 and USER2. USER1 invokes USER2 by using a LOAD and BALR sequence. The "before" part of the figure shows USER1 and USER2 residing below 16 megabytes and lists the changes necessary if USER2 moves above 16 megabytes. USER1 does not change.

The "after" part of the figure shows how things look after USER2 moves above 16 megabytes. Note that USER2 is now called USER3 and the newly created linkage assist routine has taken the name USER2.

The figure continues with a coding example that shows all three routines after the move.

BEFORE

Existing Application - USER1 invokes USER2 repeatedly



Change	Reason
• Change name of USER2 to USER3.	• USER1 does not have to change the LOAD USER2 macro.
• Write a linkage assist routine called USER2.	• USER1 remains unchanged; new USER2 switches AMODEs and branches to USER3 (the former USER).
• Change USER3 (formerly USER2) as follows:	
- Make sure all control blocks and parameters needed by USER1 and USER2 are located below the 16 megabytes line.	- USER1 and USER2 are AMODE 24; they cannot access parameters or data above 16 megabytes.
- Check mode-sensitive instructions to be sure they perform the intended function in AMODE 31, RMODE ANY.	- USER3 was moved above 16 megabytes and has the attributes AMODE 31, RMODE ANY.
- Check system services used to be sure they can be invoked in AMODE 31, RMODE ANY and make any necessary changes. (For example, change SPIE to ESPIE.) Review the Conversion Notebook chapters on incompatibilities, coexistence considerations, and programming considerations.	- USER3 has the attributes AMODE 31, RMODE ANY. SPIE and some other system services will not work in AMODE 31.
- Make sure that all fields containing addresses of areas above 16 megabytes are fullword fields.	

AFTER

Changed Application

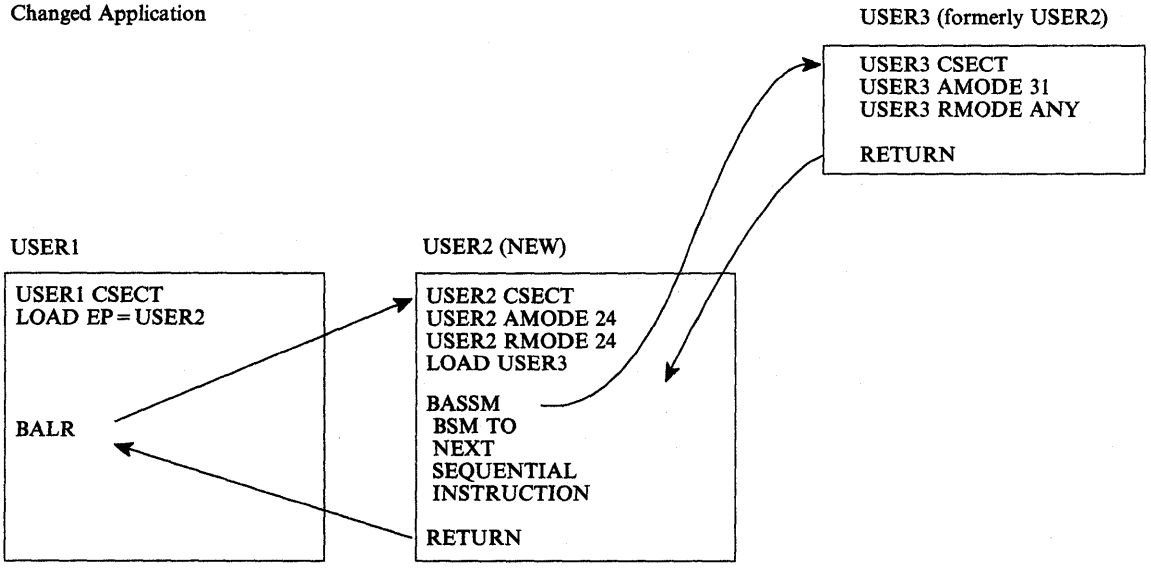


Figure 5-14 (Part 1 of 4). Example of a Linkage Assist Routine

USER1 (This module will not change)

```
*  USER MODULE USER1 CALLS MODULE USER2                          00000100
USER1    CSECT                                                   00000200
BEGIN    SAVE   (14,12),,*  (SAVE REGISTER CONTENT, ETC.)        00000300
*  ESTABLISH BASE REGISTER(S) AND NEW SAVE AREA (NORMAL          00000400
*  ENTRY CODING)                                                 00000500
           .
           .
*  ISSUE LOAD FOR MODULE USER2                                   00000700
         LOAD   EP=USER2    ISSUE LOAD FOR MODULE "USER2"       00000800
*  In the MVS/XA environment, the LOAD macro returns a
*  pointer-defined value.  However, because module USER1
*  has not been changed and executes in AMODE 24, the
*  the pointer-defined value has no effect on the BALR
*  instruction used to branch to module USER2.
         ST     0,EPUSER2   PRESERVE ENTRY POINT                00000900
           .
*  MAIN PROCESS BEGINS                                           00001000
PROCESS  DS     0H                                              00001100
           .
           .
           .
           .
           .
           .
*  PREPARE TO GO TO MODULE USER2                                 00002000
         L      15,EPUSER2  LOAD ENTRY POINT                    00002100
         BALR   14,15                                           00002200
           .
           .
           .
           .
         TM                 TEST FOR END                        00003000
         BC     PROCESS     CONTINUE IN LOOP                    00003100
           .
         DELETE EP=USER2
         L      13,4(13)
         RETURN (14,12),T,RC=0  MODULE USER1 COMPLETED          00005000
EPUSER2  DC     F'0'       ADDRESS OF ENTRY POINT TO USER2      00007000
         END    BEGIN                                           00007100
```

USER2 (Original application module)

```
*  USER MODULE USER2 (INVOKED FREQUENTLY FROM USER1)             00000100
USER2    CSECT                                                   00000200
         SAVE   (14,12),,*  SAVE REGISTER CONTENT, ETC.         00000300
*  ESTABLISH BASE REGISTER(S) AND NEW SAVE AREA (NORMAL          00000400
*  ENTRY CODING)
           .
           .
           .
           .
           .
         L      13,4(13)
         RETURN (14,12),T,RC=0  MODULE USER2 COMPLETED          00008100
         END                                                    00008200
```

Figure 5-14 (Part 2 of 4). Example of a Linkage Assist Routine

(New linkage assist routine)

```
* THIS IS A NEW LINKAGE ASSIST ROUTINE                                  0000100
* (IT WAS NAMED USER2 SO THAT MODULE USER1 WOULD NOT                    0000200
*  HAVE TO BE CHANGED)                                                  0000300
USER2    CSECT                                                         0000400
USER2    AMODE  24                                                     0000500
USER2    RMODE  24                                                     0000600
         SAVE   (14,12),,*  (SAVE REGISTER CONTENT, ETC.)              0000700
*  ESTABLISH BASE REGISTER(S) AND NEW SAVE AREA (NORMAL                 0000800
*  ENTRY CODING)
           .
* FIRST TIME LOGIC, PERFORMED ON INITIAL ENTRY ONLY,                    0002000
* (AFTER INITIAL ENTRY, BRANCH TO PROCESS (SHOWN BELOW))                0002100
           .
         GETMAIN         NEW REGISTER SAVE AREA                        0003000
           .
         LOAD   EP=USER3                                               0004000
*  USER2 LOADS USER3 BUT DOES NOT DELETE IT.  USER2 CANNOT
*  DELETE USER3 BECAUSE USER2 DOES NOT KNOW WHICH OF ITS USES
*  OF USER3 IS THE LAST ONE.
         ST     0,EPUSER3   PRESERVE POINTER DEFINED VALUE             0004100
           .
* PROCESS  (PREPARE FOR ENTRY TO PROCESSING MODULE)                     0005000
           .
         (FOR EXAMPLE, VALIDITY CHECK REGISTER CONTENTS)
           .
           .
* PRESERVE AMODE FOR USE DURING RETURN SEQUENCE                         0007000
         LA     1,XRETURN          SET RETURN ADDRESS                  0008000
         BSM    1,0                PRESERVE CURRENT AMODE              0008100
         ST     1,XSAVE            PRESERVE ADDRESS                    0008200
         L      15,EPUSER3         LOAD POINTER DEFINED VALUE          0009000
* GO TO MODULE USER3                                                    0009100
         BASSM  14,15              TO PROCESSING MODULE                0009200
* RESTORE AMODE THAT WAS IN EFFECT                                      0009300
         L      1,XSAVE            LOAD POINTER DEFINED VALUE          0009400
         BSM    0,1                SET ADDRESSING MODE                 0009500
XRETURN  DS     0H                                                     0009600
         L      13,4(13)
           .
         RETURN (14,12),T,RC=0     MODULE USER2 HAS COMPLETED          0010000
EPUSER3  DC     F'0'              POINTER DEFINED VALUE               0010100
XSAVE    DC     F'0'              ORIGINAL AMODE AT ENTRY             0010200
         END                                                           0010500
```

- Statements 8000 through 8200: These instructions preserve the AMODE in effect at the time of entry into module USER2.
- Statement 9200: This use of the BASSM instruction:
  - Causes the USER3 module to be entered in the specified AMODE (AMODE 31 in this example). This occurs because the LOAD macro returns a pointer-defined value that contains the entry point of the loaded routine, and the specified AMODE of the module.
  - Puts a pointer-defined value for use as the return address into Register 14.
- Statement 9400: Module USER3 returns to this point.
- Statement 9500: Module USER2 re-establishes the AMODE that was in effect at the time the BASSM instruction was issued (STATEMENT 9200).

Figure 5-14 (Part 3 of 4). Example of a Linkage Assist Routine

(New Application Module)

```
*  MODULE USER3    (PERFORMS FUNCTIONS OF OLD MODULE USER2)      00000100
USER3    CSECT                                                  00000200
USER3    AMODE  31                                              00000300
USER3    RMODE  ANY                                             00000400
         SAVE   (14,12),,*  (SAVE REGISTER CONTENT, ETC.)       00000500
*  ESTABLISH BASE REGISTER(S) AND NEW SAVE AREA                  00000600
            .
            .
            .
            .
            .
            .
*  RESTORE REGISTERS AND RETURN                                  00008000
            .
         RETURN (14,12),T,RC=0                                  00008100
         END                                                    00008200
```

- Statements 300 and 400 establish the AMODE and RMODE values for this module. Unless they are over-ridden by linkage editor PARM values or MODE control statements, these are the values that will be placed in the PDS for this module.
- Statement 8100 returns to the invoking module.

Figure 5-14 (Part 4 of 4). Example of a Linkage Assist Routine

Using Capping - Linkage Using a Prologue and Epilogue

An alternative to linkage assist routines is a technique called **capping**. You can add a "cap" (prologue and epilogue) to a module to handle entry and exit addressing mode switching. The cap accepts control in either 24-bit or 31-bit addressing mode, saves the caller's registers, and switches to the addressing mode in which the module is designed to run. After the module has completed its function, the epilogue portion of the cap restores the caller's registers and addressing mode before returning control.

For example, when capping is used, a module in 24-bit addressing mode can be invoked by modules whose addressing mode is either 24-bit or 31-bit; it can perform its function in 24-bit addressing mode and can return to its caller in the caller's addressing mode. Capped modules must be able to accept callers in either addressing mode. Modules that reside above 16 megabytes cannot be invoked in 24-bit addressing mode. Capping, therefore, can be done only for programs that reside below 16 megabytes.

Figure 5-15 shows a cap for a 24-bit addressing mode module.

```
MYPROG   CSECT
MYPROG   AMODE ANY
MYPROG   RMODE 24
         USING *,15
         STM 14,12,12(13) SAVE CALLER'S REGISTERS BEFORE SETTING AMODE
         LA  10,SAVE      SET FORWARD ADDRESS POINTER IN CALLER'S
         ST  10,8(13)     SAVE AREA
         LA  12,MYMODE    SET AMODE BIT TO 0 AND ESTABLISH BASE
         LA  11,RESETM    GET ADDRESS OF EXIT CODE
         BSM 11,12        SAVE CALLER'S AMODE AND SET IT TO AMODE 24
         USING *,12
MYMODE   DS    0H
         DROP 15
         ST  13,SAVE+4    SAVE CALLER'S SAVE AREA
         LR  13,10        ESTABLISH OWN SAVE AREA
-------------------------------------------------------------------------
         This is the functional part of the original module.
         This example assumes that register 11 retains its
         original contents throughout this portion of the program.
-------------------------------------------------------------------------
         L   13,4(13)     GET ADDRESS OF CALLER'S SAVE AREA
         BSM 0,11         RESET CALLER'S AMODE
RESETM   DS  0H
         LM  14,12,12(13) RESTORE CALLER'S REGISTERS IN CALLER'S AMODE
         BR  14           RETURN
  SAVE   DS  0F
         DC  18F'0'
```

Figure 5-15. Cap for an AMODE 24 Module

Performing I/O in 31-Bit Addressing Mode

Programs in 31-bit addressing mode usually need to use 24-bit addressing mode programs to perform an I/O operation because all I/O control blocks, IDAWs (indirect data address words), and CCWs must reside below 16 megabytes and all I/O requests must be made by programs executing in 24-bit addressing mode (except for VSAM). Generally, data buffers must be below 16 megabytes as well.

A 31-bit addressing mode program can perform an I/O operation by:

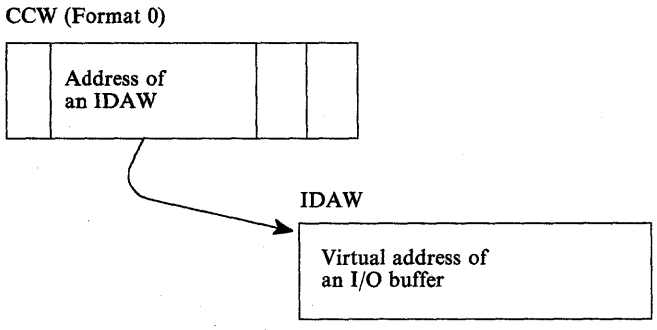
- Using VSAM services that accept callers in either 24-bit or 31-bit addressing mode. (*Managing VSAM Data Sets* describes VSAM services.)
- Using the EXCP macro. All parameter lists, control blocks, CCWs, virtual IDAWs, and EXCP appendage routines must reside in virtual storage below 16 megabytes. See "Using the EXCP Macro" for a description of using EXCP to perform I/O.
- Using the EXCPVR macro. All parameter lists, control blocks, CCWs, IDALs (indirect data address lists), and appendage routines must reside in virtual storage below 16 megabytes. See "Using EXCPVR" on page 5-39 for a description of using EXCPVR to perform I/O.
- Invoking a routine that executes in 24-bit addressing mode as an interface to non-VSAM access methods, which accept callers executing in 24-bit addressing mode only. See "Establishing Linkage" on page 5-24 for more information about this method.
- Using the method shown in Figure 5-16 on page 5-40.

To perform I/O to buffers located in virtual storage above 16 megabytes, programs must use either:

- The EXCP macro and virtual IDAWs
- The EXCPVR macro
- The VSAM access method.

Using the EXCP Macro

EXCP macro users can perform I/O to virtual storage areas above 16 megabytes. By using virtual IDAW support, CCWs in the EXCP channel program can, using a 24-bit address, point to a virtual IDAW that contains the 31-bit virtual address of an I/O buffer. The CCWs and IDAWs themselves must reside in virtual storage below 16 megabytes. The EXCP service routine supports only format 0 CCWs, the CCW format used in MVS/370.



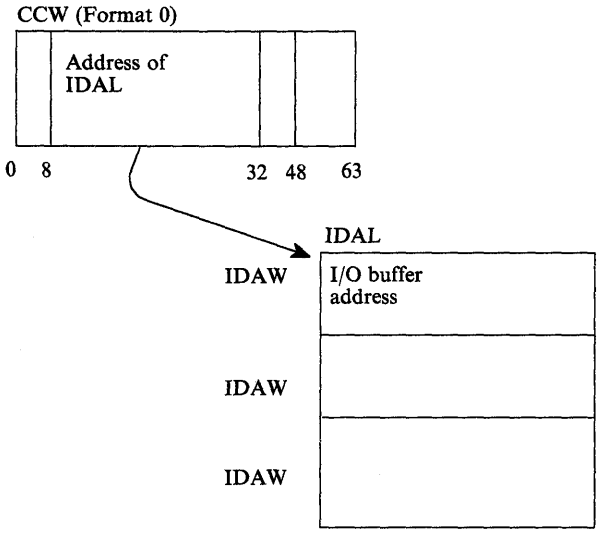
Any CCW that causes data to be transferred can point to a virtual IDAW. Virtual IDAW support is limited to non-VIO data sets.

Although the I/O buffer can be in virtual storage above 16 megabytes, the virtual IDAW that contains the pointer to the I/O buffer and all the other areas related to the I/O operation (CCWs, IOBs, DEBs, and appendages) must reside in virtual storage below 16 megabytes.

Using EXCPVR

The EXCPVR interface supports only format 0 CCWs. Format 0 CCWs support only 24-bit addresses. All CCWs and IDAWs used with EXCPVR must reside in virtual or central storage below 16 megabytes. The largest virtual or central storage address you can specify directly in your channel program is 16 megabytes minus one. However, using IDAWs (indirect data address words) you can specify any central storage address and therefore you can perform I/O to any location in real or virtual storage. EXCPVR channel programs must use IDAWs to specify buffer addresses above 16 megabytes in central storage.

The format 0 CCW may contain the address of an IDAL (indirect address list), which is a list of IDAWs (indirect data address words).



You must assume that buffers obtained by data management access methods have real storage addresses above 16 megabytes.

Example of Performing I/O While Residing Above 16 Megabytes

Figure 5-16 shows a "before" and "after" situation that involves two functions, USER1 and USER2. In the BEFORE part of the example, USER1 contains both functions and resides below 16 megabytes. In the AFTER part of the example USER1 has moved above 16 megabytes. The portion of USER1 that requests data management services has been removed and remains below 16 megabytes.

The figure includes a detailed coding example that shows both USER1 and USER2.

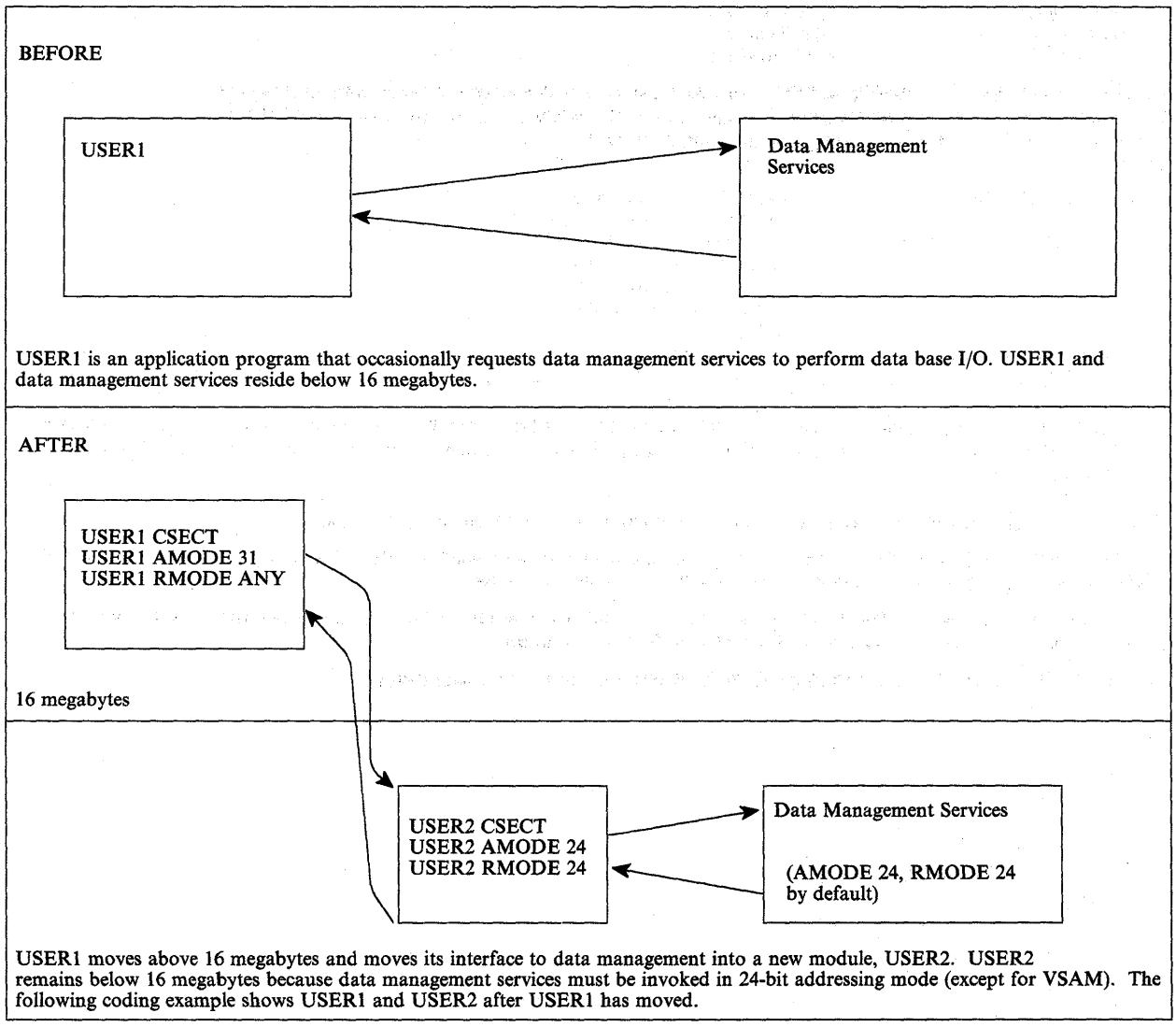


Figure 5-16 (Part 1 of 13). Performing I/O While Residing Above 16 Megabytes

USER1 Application Module

```
*Module USER1 receives control in 31-bit addressing mode, resides in
*storage above 16 megabytes, and calls module USER2 to perform data
*management services.
*In this example, note that no linkage assist routine is needed.
USER1    CSECT
USER1    AMODE 31
USER1    RMODE ANY
*
*        Save the caller's registers in save area provided
*
#100     SAVE  (14,12)              Save registers
         BASR  12,0                 Establish base
         USING *,12                 Addressability
```

Storage will be obtained via GETMAIN for USER2's work area (which will also contain the save area that module USER2 will store into as well as parameter areas in which information will be passed.) Since module USER2 must access data in the work area, the work area must be obtained below 16 megabytes.

```
         LA    0,WORKLNTH           Length of the work area
*                                   required for USER2
#200     GETMAIN RU,LV=(0),LOC=BELOW Obtain work area storage
         LR    6,1                  Save address of obtained
*                                   storage to be used for
*                                   a work area for module
*                                   USER2
         USING WORKAREA,6           Work area addressability
```

The SAVE operation at statement **#100** may save registers into a save area that exists in storage either below or above 16 megabytes. If the save area supplied by the caller of module USER1 is in storage below 16 megabytes, it is assumed that the high-order byte of register 13 is zero.

The GETMAIN at statement **#200** must request storage below 16 megabytes for the following reasons:

1. The area obtained via GETMAIN will contain the register save area in which module USER2 will save registers. Because module USER2 runs in 24-bit addressing mode, it must be able to access the save area.
2. Because module USER2 will extract data from the work area to determine what function to perform, the area must be below 16 megabytes, otherwise, USER2 would be unable to access the parameter area.

Figure 5-16 (Part 2 of 13). Performing I/O While Residing Above 16 Megabytes

```
         LA    0,GMLNTH                 Get dynamic storage for
*                                       module USER1 (USER1 resides
*                                       above 16 megabytes)
#300     GETMAIN RU,LV=(0),LOC=RES      Get storage above 16
*                                       megabytes
         LR    8,1                      Copy address of storage
*                                       obtained via GETMAIN
         USING DYNAREA,8                Base register for dynamic
*                                       work area
#400     ST    13,SAVEBKWD              Save address of caller's
*                                       save area
         LR    9,13                     Save caller's save area
*                                       address
         LA    13,SAVEAREA              USER1's save area address
*                                       Note - save area is below
*                                       16 megabytes
         ST    13,8(9)                  Have caller's save area
*                                       point to my save area
         LOAD  EP=IOSERV                Load address of data
*                                       management service
*                                       Entry point address
*                                       returned will be pointer-defined
         ST    0,EPA                    Save address of loaded
*                                       routine.
```

The GETMAIN at statement **#300** requests that the storage to be obtained match the current residency mode of module USER1. Because the module resides above 16 megabytes, the storage obtained will be above 16 megabytes.

At statement **#400**, the address of the caller's save area is saved in storage below 16 megabytes.

Figure 5-16 (Part 3 of 13). Performing I/O While Residing Above 16 Megabytes

Prepare to open input and output data base files

```
         MVC   FUNCTION,OPEN1           Indicate open file 1
*                                       for input
         LA    1,COMMAREA               Set up register 1 to
*                                       point to the parameter
*                                       area
#500     L     15,EPA                   Get pointer-defined address
*                                       of the I/O service
*                                       routine
#600     BASSM 14,15                    Call the service routine
*                                       Note: AMODE will change
#650     MVC   FUNCTION,OPEN2           Indicate open file 2
*                                       for output
         LA    1,COMMAREA               Setup register 1 to
*                                       point to the parameter
*                                       area
#700     L     15,EPA                   Get pointer-defined address
*                                       of the I/O service
*                                       routine
         BASSM 14,15                    Call the service routine
*                                       Note: AMODE will change
```

The entry point address loaded at statements **#500** and **#700** is pointer-defined, as returned by the LOAD service routine. That is, the low-order three bytes of the symbolic field EPA will contain the virtual address of the loaded routine while the high order bit (bit 0) will be zero to indicate the loaded module is to receive control in 24-bit addressing mode. The remaining bits (1-7) will also be zero in the symbolic field EPA.

The BASSM at statement **#600** does the following:

- Places into bit positions 1-31 of register 14 the address of statement **#650**.
- Sets the high-order bit of register 14 to one to indicate the current addressing mode.
- Replaces bit positions 32-63 of the current PSW with the contents of register 15 (explained above)

The BSM instruction used by the called service routine USER2 to return to USER1 will reestablish 31-bit addressing mode.

Figure 5-16 (Part 4 of 13). Performing I/O While Residing Above 16 Megabytes

Prepare to read a record from data base file 1.

```
READRTN  DS    0H
         MVC   FUNCTION,READ1          Indicate read to file 1
         XC    BUFFER,BUFFER           Clear input buffer
         LA    1,COMMAREA              Set up register 1 to
*                                      point to the parameter
*                                      area
         L     15,EPA                  Get pointer-defined address
*                                      of the I/O service
*                                      routine
         BASSM 14,15                   Call the service routine
*                                      Note: The BASSM will change
*                                      the AMODE to 24-bit.  The
*                                      BSM issued in the service
*                                      routine will reestablish
*                                      31-bit addressing mode.
#900     CLC   STATUS,ENDFILE          End of file encountered
*                                      by module USER2 ?
         BE    EODRTN                  Close files and exit
         MVC   BUFFR31A,BUFFER         Move record returned to
*                                      storage above 16
*                                      megabytes
```

At statement **#900**, a check is made to determine if called module USER2 encountered the end of file. The end of file condition in this example can only be intercepted by USER2 because the EOD exit address specified on the DCB macro must reside below 16 megabytes. The end of file condition must then be communicated to module USER1.

Figure 5-16 (Part 5 of 13). Performing I/O While Residing Above 16 Megabytes

Call a record analysis routine that exists above 16 megabytes.

```
         LA    1,BUFFR31A              Get address of first buffer
         ST    1,BUFPTR+0              Store into parameter list
         LA    1,UPDATBFR              Get address of output
*                                      buffer
         ST    1,BUFPTR+4              Store into parameter list
         LA    1,BUFPTR                Set up pointers to work
*                                      buffers for the analysis
*                                      routine
         L     15,ANALYZE              Address of analysis routine
#1000    BASR  14,15                   Call analysis routine
         MVC   BUFFER,UPDATBFR         Move updated record to
*                                      storage below 16 megabytes
*                                      so that the
*                                      updated record can
*                                      be written back to the
*                                      data base
```

At statement **#1000** a **BASR** instruction is used to call the analysis routine since no AMODE switching is required. A BALR could also have been used. A **BALR** executed while in 31-bit addressing mode performs the same function as a **BASR** instruction. The topic "Mode Sensitive Instructions" describes the instruction differences.

Figure 5-16 (Part 6 of 13). Performing I/O While Residing Above 16 Megabytes

```
         MVC   FUNCTION,WRITE1       Indicate write to file 1
         LA    1,COMMAREA            Set up register 1 to
*                                    point to the parameter
*                                    area
         L     15,EPA                Get pointer-defined address
*                                    of the I/O service
*                                    routine
         BASSM 14,15                 Call the service routine
*                                    Note: The BASSM will set
*                                    the AMODE to 24-bit.  The
*                                    BSM issued in the service
*                                    routine will reestablish
*                                    31-bit addressing mode
         B     READRTN               Get next record to
*                                    process
```

Prepare to close input and output data base files

```
EODRTN   DS    0H                    End of data routine
         MVC   FUNCTION,CLOSE1       Indicate close file 1
         LA    1,COMMAREA            Set up register 1 to
*                                    point to the parameter
*                                    area
         L     15,EPA                Get pointer-defined address
*                                    of the I/O service
*                                    routine
         BASSM 14,15                 Call the service routine
*                                    Note: The BASSM sets
*                                    the AMODE to 24-bit.  The
*                                    BSM issued in the service
*                                    routine will reestablish
*                                    31-bit addressing mode
         MVC   FUNCTION,CLOSE2       Indicate close file 2
         LA    1,COMMAREA            Set up register 1 to
*                                    point to the parameter
*                                    area
         L     15,EPA                Get pointer-defined address
*                                    of the I/O service
*                                    routine
         BASSM 14,15                 Call the service routine
*                                    Note: The BASSM sets
*                                    the AMODE to 24-bit.  The
*                                    BSM issued in the service
*                                    routine will reestablish
*                                    31-bit addressing mode
```

Figure 5-16 (Part 7 of 13). Performing I/O While Residing Above 16 Megabytes

Prepare to return to the caller

```
         L     13,SAVEBKWD             Restore save area address
*                                      of the caller of module
*                                      USER1
         LA    0,WORKLNTH              Length of work area and
*                                      parameter area used by
*                                      module USER2
         FREEMAIN RC,LV=(0),A=(6)      Free storage
         DROP  6
         LA    0,GMLNTH                Length of work area used
*                                      by USER1
         FREEMAIN RC,LV=(0),A=(8)      Free storage
         DROP  8
         XR    15,15                   Set return code zero
         RETURN (14,12),RC=(15)
```

Define DSECTs and constants for module to module communication Define constants used to communicate the function module USER2 is to perform.

```
         DS    0F
READ1    DC    C'R1'                   Read from file 1 opcode
WRITE1   DC    C'W1'                   Write to file 1 opcode
OPEN1    DC    C'O1'                   Open file 1 opcode
OPEN2    DC    C'O2'                   Open file 2 opcode
CLOSE1   DC    C'C1'                   Close file 1 opcode
CLOSE2   DC    C'C2'                   Close file 2 opcode
ANALYZE  DC    V(ANALYSIS)             Address of record
*                                      analysis routine
ENDFILE  DC    C'EF'                   End of file indicator
WORKAREA DSECT
SAVEREGS DS    0F                      This storage exists
*                                      below 16 megabytes
SAVEAREA EQU   SAVEREGS
SAVERSVD DS    F                       Reserved
SAVEBKWD DS    F
SAVEFRWD DS    F
SAVE1412 DS    15F                     Save area for registers 14-12
COMMAREA DS    0F                      Parameter area used to
*                                      communicate with module
*                                      USER2
FUNCTION DS    CL2                     Function to be performed
*                                      by USER2
STATUS   DS    CL2                     Status of read operation
BUFFER   DS    CL80                    Input/output buffer
WORKLNTH EQU   *-WORKAREA              Length of this DSECT
```

Figure 5-16 (Part 8 of 13). Performing I/O While Residing Above 16 Megabytes

Define DSECT work area for module USER1

```
DYNAREA  DSECT                         This storage exists
*                                      above 16 megabytes
EPA      DS    F                       Address of loaded routine
BUFFR31A DS    CL80                    Buffer - above 16
*                                      megabytes
BUFPTR   DS    0F
         DS    A                       Address of input buffer
         DS    A                       Address of updated buffer
UPDATBFR DS    CL80                    Updated buffer returned
*                                      by the analysis routine
GMLNTH   EQU   *-DYNAREA               Length of dynamic area
         END
```

Figure 5-16 (Part 9 of 13). Performing I/O While Residing Above 16 Megabytes

USER2 Service Routine

```
*Module USER2 receives control in 24-bit addressing mode, resides below
*16 megabytes, and is called by module USER1 to perform data
*management services.

USER2    CSECT
USER2    AMODE 24
USER2    RMODE 24
*
*        Save the caller's registers in save area provided
*
         SAVE  (14,12)              Save registers
         BASR  12,0                 Establish base
         USING *,12                 Addressability
         LR    10,1                 Save parameter area pointer
*                                   around GETMAINs
         USING  COMMAREA,10         Establish parameter area
*                                   addressability
```

Storage will be obtained via GETMAIN for a save area that module USER2 can pass to external routines it calls.

```
         LA    0,WORKLNTH           Length of the work area
*                                   required
         GETMAIN RU,LV=(0),LOC=RES  Obtain save area storage,
*                                   which must be below
*                                   16 megabytes
         LR    6,1                  Save address of obtained
*                                   storage to be used for
*                                   a save area for module
*                                   USER2
         USING SAVEREGS,6           Save area addressability
         LA    0,GMLNTH             Get dynamic storage for
*                                   module USER2 below
*                                   16 megabytes.
```

Note:This GETMAIN will only be done on the initial call to this module.

```
#2000    GETMAIN RU,LV=(0),LOC=RES  Obtain storage below
*                                   16 megabytes
         LR    8,1                  Copy address of storage
*                                   obtained via GETMAIN
         USING DYNAREA,8            Base register for the
*                                   dynamic work area
         ST    13,SAVEBKWD          Save address of caller's
*                                   save area
         LR    9,13                 Save caller's save area
*                                   address
         LA    13,SAVEAREA          USER1's save  area address
*                                   Note - save area is
*                                   below 16 megabytes
```

The GETMAIN at statement **#2000** requests that storage be obtained to match the current residency mode of module USER2. Because the module resides below 16 megabytes, storage obtained will be below 16 megabytes.

Figure 5-16 (Part 10 of 13). Performing I/O While Residing Above 16 Megabytes

Note: The following store operation is successful because module USER1 obtained save area storage below 16 megabytes.

```
         ST    13,8(9)                 Have caller's save area
*                                      point to my save area
         .
         .
         .
         .
* Process input requests
         .
         .
         .
         .
READ1    DS    0H                      Read a record routine
         .
         L     13,SAVEBKWD
         LM    14,12,12(13)            Reload USER1's registers
         BSM   0,14                    Return to caller - this
*                                      instruction sets AMODE 31
WRITE1   DS    0H                      Write a record routine
         .
         L     13,SAVEBKWD
         LM    14,12,12(13)            Reload USER1's registers
         BSM   0,14                    Return to caller - this
*                                      instruction sets AMODE 31
OPEN1    DS    0H                      Open file 1 for input
         .
         L     13,SAVEBKWD
         LM    14,12,12(13)            Restore caller's registers
         BSM   0,14                    Return to caller - this
*                                      instruction sets AMODE 31
CLOSE1   DS    0H                      Close file 1 for input
         .
         L     13,SAVEBKWD
         LM    14,12,12(13)            Restore caller's registers
         BSM   0,14                    Return to caller - this
*                                      instruction sets AMODE 31
OPEN2    DS    0H                      Open file 2 for input
         .
         L     13,SAVEBKWD
         LM    14,12,12(13)            Restore caller's registers
         BSM   0,14                    Return to caller - this
*                                      instruction sets AMODE 31
CLOSE2   DS    0H                      Close file 2 for input
         .
         L     13.SAVEBKWD
         LM    14,12,12(13)            Restore caller's registers
         BSM   0,14                    Return to caller - this
*                                      instruction sets AMODE 31
         .
         .
```

Figure 5-16 (Part 11 of 13). Performing I/O While Residing Above 16 Megabytes

Note:This FREEMAIN will only be done on the final call to this module.

```
         LA    0,GMLNTH              Length of work area used
*                                    by USER2
         FREEMAIN RC,LV=(0),A=(8)    Free storage
         .
         .
         .
```

DCB END OF FILE and ERROR ANALYSIS ROUTINES

```
ENDFILE  DS    0H                    End of file encountered
         .
         .
         MVC   STATUS,ENDFILE        Indicate end of file to
*                                    module USER1
         .
         L     13,SAVWBKWD
         LM    14,12,12(13)          Reload USER1's registers
         BSM   0,14                  Return to USER1
*                                    indicating end of file
*                                    has been encountered
         .
         .
         .
         .
ERREXIT1 DS    0H                    SYNAD error exit one
         .
         .
         MVC   STATUS,IOERROR        Indicate I/O error to
*                                    module 'USER1'
         .
         L     13,SAVWBKWD
         LM    14,12,12(13)          Reload USER1's registers
         BSM   0,14                  Return to USER1
*                                    indicating an I/O error
*                                    has been encountered
         .
         .
         .
         .
ERREXIT2 DS    0H                    SYNAD error exit two
         .
         .
         MVC   STATUS,IOERROR        Indicate I/O error to
*                                    module 'USER1'
         .
         L     13,SAVWBKWD
         LM    14,12,12(13)          Reload USER1's registers
         BSM   0,14                  Return to USER1
*                                    indicating an I/O error
*                                    has been encountered
```

Figure 5-16 (Part 12 of 13). Performing I/O While Residing Above 16 Megabytes

Note:Define the required DCBs that module USER2 will process. The DCBs exist in storage below 16 megabytes. The END OF DATA and SYNAD exit routines also exist in storage below 16 megabytes.

```
INDCB    DCB   DDNAME=INPUT1,DSORG=PS,MACRF=(GL),EODAD=ENDFILE, x
               SYNAD=ERREXIT1
OUTDCB   DCB  DDNAME=OUTPUT1,DSORG=PS,MACRF=(PL),SYNAD=ERREXIT2
* Work areas and constants for module USER2
IOERROR  DC    C'IO'                  Constant used to indicate
*                                     an I/O error
ENDFILE  DC    C'EF'                  Constant used to indicate
*                                     end of file encountered
SAVEREGS DSECT                        This storage exists
*                                     below 16 megabytes
SAVEAREA EQU   SAVEREGS
SAVERSVD DS    F                      Reserved
SAVEBKWD DS    F
SAVEFRWD DS    F
SAVE1412 DS    15F                    Save area for registers 14-12
WORKLNTH EQU   *-SAVEREGS             Length of dynamic area
         .
         .
         .
         .
         .
COMMAREA DSECT                        Parameter area used to
*                                     communicate with module
*                                     USER1
FUNCTION DS    CL2                    Function to be performed
*                                     by USER2
STATUS   DS    CL2                    Status of read operation
BUFFER   DS    CL80                   Input/output buffer
         .
         .
DYNAREA  DSECT                        This storage exists
*                                     below 16 megabytes
         .
         .
         .
         .
         .
         .
GMLNTH   EQU   *-DYNAREA              Length of dynamic area
         .
         .
         END
```

Figure 5-16 (Part 13 of 13). Performing I/O While Residing Above 16 Megabytes

Understanding the Use of Central Storage

MVS programs and data reside in virtual storage that, when necessary, is backed by central storage. Most programs and data do not depend on their real addresses. Some MVS programs, however, do depend on real addresses and some require these real addresses to be less than 16 megabytes. MVS reserves as much central storage below 16 megabytes as it can for such programs and, for the most part, handles their central storage dependencies without requiring them to make any changes.

The system uses the area of central storage above 16 megabytes to back virtual storage with real frames whenever it can. All virtual areas above 16 megabytes can be backed with real frames above 16 megabytes. In addition, the following virtual areas below 16 megabytes can also be backed with real frames above 16 megabytes:

- SQA
- LSQA
- Nucleus
- Pageable private areas
- Pageable CSA
- PLPA
- MLPA
- Resident BLDL

The following virtual areas are always backed with real frames below 16 megabytes:

- V=R regions
- FLPA
- Subpool 226
- Subpools 227 and 228 (unless specified otherwise by the GETMAIN macro with the LOC parameter)

When satisfying page-fix requests, MVS backs pageable virtual pages that reside below 16 megabytes with central storage below 16 megabytes, unless the GETMAIN macro specifies LOC=(BELOW,ANY) or the PGSER macro specifies the ANYWHER parameter. If the GETMAIN or STORAGE macro specifies or implies a real location of ANY, MVS/XA backs pageable virtual pages with real frames above 16 megabytes even when the area is page fixed.

Central Storage Considerations for User Programs

Among the things to think about when dealing with central storage in 31-bit addressing mode are the use of the load real address (LRA) instruction, the use of the LOC parameter on the GETMAIN macro, the location of the DAT-off portion of the nucleus, and using EXCPVR to perform I/O. (EXCPVR was described in the section Performing I/O in 31-Bit Addressing Mode.)

Load Real Address (LRA) Instruction

The LRA instruction always results in a 31-bit real address regardless of the issuing program's addressing mode. All programs that issue an LRA instruction must be prepared to handle a 31-bit result if the virtual storage address specified could have been backed with central storage above 16 megabytes. Issue LRA only against areas that are fixed.

GETMAIN Macro

The LOC parameter on the RU, RC, VRU, and VRC forms of the GETMAIN macro specifies not only the virtual storage location of the area to be obtained, but also the central storage location when the storage is page fixed.

LOC=BELOW indicates that the virtual storage is to be located below 16 megabytes. When the area is page fixed, it is to be backed with central storage below 16 megabytes.

LOC=(BELOW,ANY) indicates that virtual storage is to be located below 16 megabytes but that central storage can be located either above or below 16 megabytes.

LOC=ANY and LOC=(ANY,ANY) indicate that both virtual and central storage can be located either above or below 16 megabytes.

LOC=RES indicates that the location of virtual and central storage depends on the location (RMODE) of the GETMAIN issuer. If the issuer has an RMODE of 24, LOC=RES indicates the same thing as LOC=BELOW; if the issuer has an RMODE of ANY, LOC=RES indicates the same thing as LOC=ANY.

LOC=(RES,ANY) indicates that the location of virtual storage depends on the location of the issuer but that central storage can be located anywhere.

LOC is optional except in the following case: A program that resides above 16 megabytes and uses the RU, RC, VRU, and VRC forms of GETMAIN **must** specify either LOC=BELOW or LOC=(BELOW,ANY) on GETMAIN requests for storage that will be used by programs running in 24-bit addressing mode. Otherwise, virtual storage would be assigned above 16 megabytes and 24-bit addressing mode programs could not use it.

The location of virtual storage can also be specified on the PGSER (page services) macro. The ANYWHER parameter on PGSER specifies that a particular virtual storage area can be placed either above or below 16 megabytes on future page-ins. This parameter applies to virtual storage areas where LOC=(BELOW,ANY) or LOC=(ANY,ANY) was not specified on GETMAIN.

DAT-Off Routines

The MVS/370 nucleus is mapped so that its virtual storage addresses are equal to its central storage addresses. MVS/370 has a V=R (virtual=real) nucleus. In contrast, the MVS/XA and MVS/ESA nucleus is mapped and fixed in central storage without attempting to make its virtual storage addresses equal to its real addresses. MVS systems that use 31-bit addressing (MVS/XA and MVS/ESA) have a V=F (virtual=fixed) nucleus.

Because the MVS/370 is V=R, nucleus code can turn DAT off, and the next instruction executed is the same as it would be if DAT was still on. Because the MVS/XA and MVS/ESA nucleus is not V=R, MVS/XA and MVS/ESA nucleus code cannot turn DAT-off and expect the next instruction executed to be the same as if DAT was on.

To allow for the execution of DAT-off nucleus code, the MVS/XA nucleus consists of two load modules, one that runs with DAT on and one that runs with DAT off. Nucleus code that needs to run with DAT off must reside in the DAT-off portion of the nucleus.

When the system is initialized, the DAT-off portion of the nucleus is loaded into the highest contiguous central storage. Therefore, you must modify any user modules in the nucleus that run with DAT off so that they operate correctly above 16 megabytes. Among the things you may have to consider are:

- All modules in the DAT-off portion of the nucleus have the attributes AMODE 31, RMODE ANY. They may reside above 16 megabytes.
- These modules must return control via a BSM 0,14.
- Register 0 must not be destroyed on return.

To support modules in the DAT-off nucleus:

- Move the DAT-off code to a separate module with AMODE 31, RMODE ANY attributes. Use as its entry point, IEAVEURn where n is a number from 1 to 4. (MVS/XA reserves four entry points in the DAT-off nucleus for users.) Use BSM 0,14 as the return instruction. Do not destroy register 0.
- Code a DATOFF macro to invoke the DAT-off module:

  ```
  DATOFF INDEX=INDUSRn
  ```

  The value of n in INDUSRn must be the same as the value of n in IEAVEURn, the DAT-off module's entry point.
- Link edit the DAT-off module (IEAVEURn) into the IEAVEDAT member of SYS1.NUCLEUS (the DAT-off nucleus).

Refer to *Authorized Assembler Programming Guide* and *Authorized Assembler Programming Reference* for more information about modifying the DAT-off portion of the nucleus and the DATOFF macro.

Chapter 6. Resource Control

When your program executes, other programs are executing concurrently in the MVS multiprogramming environment. Each group of programs, including yours, is a competitor for resources available at execution time. A resource is anything that a program needs as it executes — such as processor time, a data set, another program, a table, or a hardware device, etc. The competitor for resources is actually the *task* that represents the program.

If you subdivide a program into separate logical parts, and code it as several small programs instead of one large program, you can make the parts execute as separate tasks and with greater efficiency. However, you must ensure that each part executes in correct order relative to the others.

This chapter describes how your program can run in an environment in which it must share resources with other programs.

- The WAIT, POST, and EVENTS macros introduce a strategic delay in the running of a program. This delay forces a program to wait for a particular event to occur. When the event occurs, the program can run once again. The event can be the availability of a necessary resource.
- The ENQ and DEQ macros allow many programs to serialize use of resources, such as data sets. The sharing of resources often requires that a program, or many programs, enter a wait state until a requested resource becomes available.

The macros described in this chapter are available only to programs that are running in task mode; programs represented by service request blocks (SRBs) must use other methods of controlling the use of resources.

Synchronizing Tasks (WAIT, POST, and EVENTS Macros)

Some planning on your part is required to determine what portions of one task are dependent on the completion of portions of all other tasks. The POST macro is used to signal completion of an event; the WAIT and EVENTS macros are used to indicate that a task cannot proceed until one or more events have occurred. An event control block (ECB) is used with the WAIT, EVENTS or POST macros; it is a fullword on a fullword boundary, as shown in Figure 6-1.

An ECB is also used when the ECB parameter is coded in an ATTACH or ATTACHX macro. In this case, the control program issues the POST macro for the event (subtask termination). Either the 24-bit (bits 8 to 31) return code in register 15 (if the task completed normally) or the completion code specified in the ABEND macro (if the task was abnormally terminated) is placed in the ECB as shown in Figure 6-1. The originating task can issue a WAIT macro or EVENTS macro with WAIT=YES parameter specifying the ECB; the task will not regain control until after the event has taken place and the ECB is posted (except if an asynchronous event occurs, for example, timer expiration).

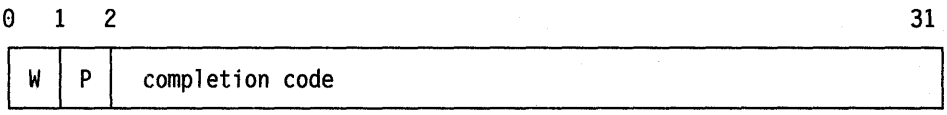


Figure 6-1. Event Control Block (ECB)

When an ECB is originally created, bits 0 (wait bit) and 1 (post bit) must be set to zero. If an ECB is reused, bits 0 and 1 must be set to zero before a WAIT, EVENTS ECB= or POST macro can be specified. If, however, the bits are set to zero before the ECB has been posted, any task waiting for that ECB to be posted will remain in the wait state. When a WAIT macro is issued, bit 0 of the associated ECB is set to 1. When a POST macro is issued, bit 1 of the associated ECB is set to 1 and bit 0 is set to 0. For an EVENTS type ECB, POST also puts the completed ECB address in the EVENTS table.

A WAIT macro can specify more than one event by specifying more than one ECB. (Only one WAIT macro can refer to an ECB at a time, however.) If more than one ECB is specified in a WAIT macro, the WAIT macro can also specify that all or only some of the events must occur before the task is taken out of the wait condition. When a sufficient number of events have taken place (ECBs have been posted) to satisfy the number of events indicated in the WAIT macro, the task is taken out of the wait condition.

An optional parameter, LONG=YES or NO, allows you to indicate whether the task is entering a long wait or a regular wait. A long wait should never be considered for I/O activity. However, you might want to use a long wait when waiting for an operator response to a WTOR macro.

Through the LINKAGE parameter, POST and WAIT allow you to specify how the macro passes control to the control program. You can specify that control is to be passed by an SVC or a PC instruction.

When you issue the WAIT or POST macro and specify LINKAGE=SVC (or use the default), your program must not be in cross memory mode. The primary, secondary, and home address spaces must be the same, your program must be in primary ASC mode, and it must not have an enabled unlocked task (EUT) functional recovery routine (FRR) established. You may use WAIT and POST when the primary and the home address spaces are different by specifying LINKAGE=SYSTEM. This option generates a PC interface to the WAIT or POST service and requires that the program be enabled, unlocked, in primary ASC mode and, for WAIT only, in task mode. For POST, the control program assumes that the ECB is in the primary address space. For WAIT, it assumes that the ECB is in the home address space.

Figure 6-2 on page 6-3 shows an example of using LINKAGE=SYSTEM. The program that runs under TCB1 in ASN1 PCs to a program in ASN2. Now the primary address space is ASN2 and home address space is ASN1. When the PC routine POSTs ECB2, it uses LINKAGE=SYSTEM because home and primary are different. The PC routine WAITs on ECB1 using LINKAGE=SYSTEM because home and primary are still different. Note that ECB1 is in the home address space.

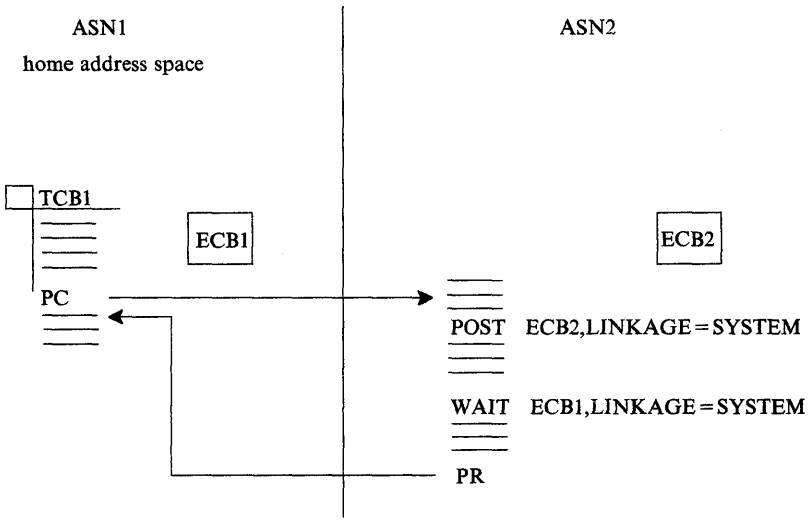


Figure 6-2. Using LINKAGE=SYSTEM on the WAIT and POST Macros

Serializing Access to Resources (ENQ and DEQ Macros)

When one or more programs using a serially reusable resource modify the resource, they must not use the resource simultaneously with other programs. Consider a data area in virtual storage that is being used by programs associated with several tasks of a job step. Some of the programs are only reading records in the data area; because they are not updating the records, they can access the data area simultaneously. Other programs using the data area, however, are reading, updating, and replacing records in the data area. Each of these programs must serially acquire, update, and replace records by locking out other programs. In addition, none of the programs that are only reading the records want to use a record that another program is updating until after the record has been replaced.

If your program uses a serially reusable resource, you must prevent incorrect use of the resource. You must ensure that the logic of your program does not require the second use of the resource before completion of the first use. Be especially careful when using a serially reusable resource in an exit routine; because exit routines get control asynchronously with respect to your program logic, the exit routine could obtain a resource already in use by the main program. When more than one task is involved, using the ENQ and DEQ macros correctly can prevent simultaneous use of a serially reusable resource.

The ENQ macro assigns control of a resource to the current task. The control program determines the status of the resource and does one of the following:

- If the resource is available, the control program grants the request by returning control to the active task.
- If the resource has been assigned to another task, the control program delays assignment of control by placing the active task in a wait condition until the resource becomes available.
- Passes back to the caller on conditional requests a return code indicating the status of the resource.
- Abends the caller on unconditional requests that would otherwise cause a non-zero return code.

When the status of the resource changes so that the waiting task can get control, the task is taken out of the wait condition and placed in the ready condition.

The ENQ and DEQ macros work together to protect serially reusable resources. The rules for proper use of ENQ/DEQ are as follows:

- Everyone must use ENQ/DEQ.
- Everyone must use the same names and scope values for the same resources.
- Everyone must use consistent ENQ/DEQ protocol.

Naming the Resource

The ENQ and DEQ macros identify the resource by two names known as the qname and the rname, and by a scope value. The qname and rname need not have any relation to any actual name of the resource. The control program does not associate a name with an actual resource; it merely processes requests having the same qname, rname, and scope on a first-in, first-out basis. It is up to you to associate the names with the resource by ensuring that all users of the resource use the same qname, rname, and scope value to represent the same resource. The control program treats requests having different qname, rname, and scope combinations as requests for different resources. Because the control program cannot determine the actual name of the resource from the qname, rname, and scope, a task could use the resource by specifying a different qname, rname, and scope combination or by accessing the resource without using ENQ. In this case, the control program cannot provide any protection.

Choose qnames and rnames carefully. Because the control program uses SYSZ for its qnames, the task abends if you use SYSZ as the first four characters of a qname. Avoid using SYSA through SYSY because the control program sometimes uses these characters for its qnames as well. Either check with your system programmer to see which of the SYSA through SYSY combinations you can use or avoid using SYSx (where x is alphabetic) to begin qnames.

Defining the Scope of a Resource

You can request a scope of STEP, SYSTEM, or SYSTEMS on the ENQ and DEQ macros.

- Use a scope of STEP if the resource is used only in your address space. The control program uses the address space identifier to make your resource unique in case someone else in another address space uses the same qname and rname and a scope of STEP.
- Use a scope of SYSTEM if the resource is available to more than one address space in the same system. All programs on that system that serialize on the resource must use the same qname and rname and a scope of SYSTEM. For example, to prevent two jobs from using a named resource simultaneously, use SYSTEM.
- Use a scope of SYSTEMS if the resource is available to more than one system. All programs that serialize on the resource must use the same qname and rname and a scope of SYSTEMS. For example, to prevent two processors from using a named resource simultaneously, use SYSTEMS. Note that the control program considers a resource with a SYSTEMS scope to be different from a resource represented by the same qname and rname but with a scope of STEP or SYSTEM.

Local and Global Resources

The ENQ and DEQ macros recognize two types of resources: local resources and global resources.

A local resource is a resource identified on ENQ or DEQ by a scope of STEP or SYSTEM. A local resource is recognized and serialized only within the requesting operating system. The local resource queues are updated to reflect each request for a local resource. If a system is not operating under global resource serialization (that is, the system is not part of a global resource serialization complex), all resources requested are treated as local resources.

If a system is part of a global resource serialization complex, a global resource is identified on the ENQ or DEQ macro by a scope of SYSTEMS. A global resource is recognized and serialized by all systems in the global resource serialization complex.

If your system is part of a global resource serialization complex, global resource serialization might change the scope value during its resource name list (RNL) processing. If the resource appears in the SYSTEM inclusion RNL, a resource with a scope of SYSTEM can have its scope converted to SYSTEMS. If the resource appears in the SYSTEMS exclusion RNL, a scope of SYSTEMS can have its scope changed to SYSTEM. This important procedure is described in *Planning: Global Resource Serialization*.

Through the RNL parameter on ENQ and DEQ, you can request that global resource serialization not perform RNL processing and, therefore, not change the scope value of a resource. It is recommended that you use RNL=YES, the default, which tells global resource serialization to perform RNL processing. Use RNL=NO only when you are sure that you **never** want the scope value to change. An example of the use of RNL=NO is in a cross-system coupling facility (XCF) complex, where you can be certain that certain data sets always need a scope value of SYSTEMS and other data sets always need a scope value of SYSTEM. In a sense, RNL=NO overrides decisions your system programmer makes when that programmer places resource names in the RNLs.

Because the RNL parameter affects the scope value of a resource, be consistent in specifying the RNL parameter on both ENQ and DEQ. If you use the default value on ENQ, use the default value also on DEQ.

Requesting Exclusive or Shared Control

On ENQ and DEQ, you specify either exclusive or shared control of a resource.

To request exclusive control of the resource, code E on ENQ. When your program has exclusive control of a resource, it is the only one that can use that resource; all other programs that issue ENQs for the resource (either for exclusive or shared control) must wait until your program issues DEQ to release the resource. If your program will change the resource, it must request exclusive control.

To request shared control of the resource, code S on the ENQ macro. At the same time your program has access to the resource, other programs can have concurrent use of the resource. If another program requests exclusive control over the resource during the time your program has shared use of the resource, that program will have to wait until all the current users have issued DEQ to release the resource. If your program will not change the resource, it should request shared control.

For an example of how the control program processes requests for exclusive and shared control of a resource, see "Processing the Requests" on page 6-6.

Limiting Concurrent Requests for Resources

To prevent any one job, started task, or TSO user from generating too many concurrent requests for resources, the control program counts and limits the number of ENQs in each address space. When a user issues an ENQ, the control program increases the count of outstanding requests for that address space by one and decreases the count by one when the user issues a DEQ.

When the computed count reaches the threshold value or limit, the control program processes subsequent requests as follows:

- Unconditional requests (ENQs that use the RET=NONE option) are abended with a system code of X'538'.
- Conditional requests (ENQs that specify the RET=HAVE or RET=USE option) are rejected and the user receives a return code of X'18'.

The RESERVE and GQSCAN macros, available only to authorized programs, also increase the count of outstanding requests.

Processing the Requests

The control program constructs a unique list for each qname, rname, and scope combination it receives. When a task makes a request, the control program searches the existing lists for a matching qname, rname, and scope. If it finds a match, the control program adds the task's request to the end of the existing list; the list is not ordered by the priority of the tasks on it. If the control program does not find a match, it creates a new list, and adds the task's request as the first (and only) element. The task gets control of the resource based on the following:

- The position of the task's request on the list
- Whether or not the request was for exclusive or shared control

The best way to describe how the control program processes the list of requests for a resource is through an example. Figure 6-3 shows the status of a list built for a qname, rname, and scope combination. The S or E next to the entry indicates that the request was for either shared or exclusive control. The task represented by the first entry on the list always gets control of the resource, so the task represented by ENTRY1 (Figure 6-3, Step 1) is assigned the resource. The request that established ENTRY2 was for exclusive control, so the corresponding task is placed in the wait condition, along with the tasks represented by all the other entries in the list.

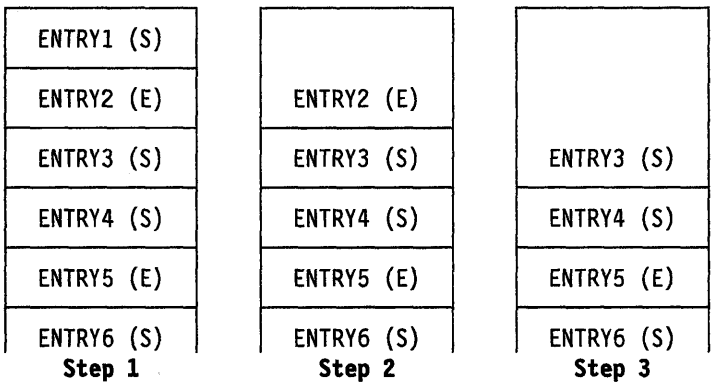


Figure 6-3. ENQ Macro Processing

Eventually, the task represented by ENTRY1 uses DEQ to release control of the resource, and the ENTRY1 is removed from the list. As shown in Figure 6-3, Step 2, ENTRY2 is now first on the list, and the corresponding task is assigned control of the resource. Because the request that established ENTRY2 was for exclusive control, the tasks represented by all the other entries in the list remain in the wait condition.

Figure 6-3, Step 3, shows the status of the list after the task represented by ENTRY2 releases the resource. Because ENTRY3 is now at the top of the list, the task represented by ENTRY3 gets control of the resource. ENTRY3 indicates that the resource can be shared, and, because ENTRY4 also indicates that the resource can be shared, ENTRY4 also gets control of the resource. In this case, the task represented by ENTRY5 does not get control of the resource until both the tasks represented by ENTRY3 and ENTRY4 release control because ENTRY5 indicates exclusive use.

The control program uses the following general rules in manipulating the lists:

- The task represented by the first entry in the list always gets control of the resource.
- If the request is for exclusive control, the task is not given control of the resource until its request is the first entry in the list.
- If the request is for shared control, the task is given control either when its request is first in the list or when all the entries before it in the list also indicate a shared request.
- If the request is for several resources, the task is given control when all of the entries requesting exclusive control are first in their respective lists and all the entries requesting shared control are either first in their respective lists or are preceded only by entries requesting shared control.

Duplicate Requests for a Resource

A duplicate request occurs when a task issues an ENQ macro to request a resource that the task already controls. For example, if a task that has control of a resource issues an unconditional ENQ macro to request the same resource, the task abends. With the second request, the control program recognizes the contradiction and returns control to the task with a non-zero return code or abnormally terminates the task. You should design your program to ensure that a second request for a resource made by the same task is never issued until control of the resource is released for the first use. Be especially careful when using an ENQ macro in an exit

routine. Two specific reasons why the use of ENQ in an exit routine must be carefully planned are:

- The exit may be entered more than once for the same task.
- An exit routine may request resources already obtained by some other process associated with the task.

For more information on this topic, see "Conditional and Unconditional Requests."

Releasing the Resource

Use the DEQ macro to release a serially reusable resource that you obtained by using an ENQ macro. If a task tries to release a resource which it does not control, the control program either returns a non-zero return code or abends the task. The control program might place many tasks in a wait condition while it assigns control of the resource to one task. Having many tasks in the wait state might reduce the amount of work being done by the system, therefore, you should issue a DEQ macro as soon as possible to release the resource, so that other tasks can use it. If a task terminates without releasing a resource, the control program releases the resource automatically.

Conditional and Unconditional Requests

Up to this point, only unconditional requests have been considered. You can, however, use the ENQ and DEQ macros to make conditional requests by using the RET parameter. One reason for making a conditional request is to avoid the abnormal termination that occurs if you issue two ENQ macros for the same resource within the same task or when a DEQ macro is issued for a resource for which you do not have control.

The RET parameter of ENQ provides the following options:

RET=TEST indicates the availability of the resource is to be tested, but control of the resource is not requested.

RET=USE indicates control of the resource is to be assigned to the active task only if the resource is immediately available. If any of the resources are not available, the active task is not placed in a wait condition.

RET=CHNG indicates the status of the resource specified is changed from shared to exclusive control.

RET=HAVE indicates that control of the resource is requested conditionally; that is, control is requested only if a request has not been made previously for the same task.

For the following descriptions, the term "active task" means the task issuing the ENQ macro. No reference is intended to different tasks which might be active in other processors of a multiprocessor.

Use RET=TEST to test the status of the corresponding qname, rname, and scope combination, without changing the list in any way or waiting for the resource.

- A return code of 0 indicates that the active task does not now have control of the resource, but could have been given immediate control if it had been requested, because no other task has control of the resource.
- A return code of 4 indicates that another task has control of the resource, and the active task would have been placed in a wait condition if it had made an unconditional request.

- A return code of 8 indicates that the active task already has control of the resource.
- A return code of 14 indicates that the active task does not yet have control of the resource, but is in the list to be given control at a later time when other task(s) release the resource.

  Note: For return code 14 to occur, the restricted use of the ECB parameter of the ENQ must have been used to make an entry on the list without placing the task in a wait condition.

RET=TEST is most useful for determining if the task already has control of the resource. It is less useful for determining the status of the list and taking action based on that status. In the interval between the time the control program checks the status and the time your program checks the return code and issues another ENQ macro, another task could have been made active, and the status of the list could have changed.

Use RET=USE if you want your task to receive control of the resource only if the resource is immediately available. If the resource is not immediately available, no entry will be made on the list and the task will not be made to wait. RET=USE is most useful when there is other processing that can be done without using the resource. For example, by issuing a preliminary ENQ with RET=USE in an interactive task, you can attempt to gain control of a needed resource without locking your terminal session. If the resource is not available, you can do other work rather than enter a long wait for the resource.

- A return code of 0 indicates that the active task did not have control of the resource prior to issuing the ENQ, but now has been given control and the corresponding entry has been put in the list.
- A return code of 4 indicates that the active task has not been given control of the resource, and an entry has not been made in the list, because another task already has control of the resource.
- A return code of 8 indicates that the active task already has control of the resource.
- A return code of 14 indicates that the active task does not yet have control of the resource, but is in the list to be given control at a later time when other task(s) release the resource.
- A return code of 18 indicates that the limit for the number of concurrent resource requests has been reached. The task does not have control of the resource unless some previous ENQ/RESERVE request caused the task to obtain control of the resource.

Use RET=CHNG to change a previous request from shared to exclusive control.

- A return code of 0 indicates that the active task now has exclusive control of the resource. Either exclusive control was already held, or shared control was converted to exclusive control as requested.
- A return code of 4 indicates that the requested change in attribute cannot be honored, because the active task is currently sharing the resource with another task.
- A return code of 8 indicates that the active task does not have an entry on the list for the specified resource. There is nothing to change.

- A return code of 14 indicates that the active task does have an entry on the list for the resource, but is not yet in control of the resource. No change is made.

Use RET=HAVE to specify a conditional request for control of a resource when you do not know whether you have already requested control of that resource. If the resource is owned by another task, you will be put in a wait condition until the resource becomes available.

- A return code of 0 indicates that the active task did not previously have an entry on the list or control of the resource, but has now been given control.
- A return code of 8 indicates that the active task already has control of the resource and already has an entry on the list. (Without RET=HAVE, this situation would cause abnormal termination. With RET=HAVE, it is effectively a no-operation.)
- A return code of 14 indicates that the active task has entry on the list for the resource, but is not yet in control of the resource. No change is made.

The RET=HAVE parameter on DEQ allows you to release control and prevents the control program from abending the active task if a DEQ attempts to release a resource not held. For DEQ, the control program returns the following codes:

- A return code of 0 indicates that the DEQ routine found an entry for the active task on the list for the specified resource, and has removed the entry. If the active task held control of the resource, this action relinquishes control. If the active task did not hold control of the resource (because the restricted ECB parameter had been used with ENQ, and control has not meanwhile become available), the DEQ routine simply removes the entry from the list without affecting control of the resource.
- A return code of 4 indicates the resource has been requested for the task, but the task has not been assigned control. The task is not removed from the wait condition. (This return code could result if DEQ is issued within an exit routine which was given control because of an interruption).
- A return code of 8 indicates that the active task did not have an entry on the list for the specified resource. There was no entry to dequeue.

If ENQ and DEQ are used in an asynchronous exit routine, code RET=HAVE to avoid possible abnormal termination.

Avoiding Interlock

An interlock condition happens when two tasks are waiting for each other's completion, but neither task can get the resource it needs to complete. Figure 6-4 shows an example of an interlock. Task A has exclusive access to resource M, and task B has exclusive access to resource N. When task B requests exclusive access to resource M, B is placed in a wait state because task A has exclusive control of resource M.

The interlock becomes complete when task A requests exclusive control of resource N. The same interlock would have occurred if task B issued a single request for multiple resources M and N prior to task A's second request. The interlock would not have occurred if both tasks had issued single requests for multiple resources. Other tasks requiring either of the resources are also in a wait condition because of the interlock, although in this case they did not contribute to the conditions that caused the interlock.

```
      Task A                  Task B
ENQ (M,A,E,8,SYSTEM)
                        ENQ (N,B,E,8,SYSTEM)
                        ENQ (M,A,E,8,SYSTEM)
ENQ (N,B,E,8,SYSTEM)
```

Figure 6-4. Interlock Condition

The above example involving two tasks and two resources is a simple example of an interlock. The example could be expanded to cover many tasks and many resources. It is imperative that you avoid interlock. The following procedures indicate some ways of preventing interlocks.

- Do not request resources that you do not need immediately. If you can use the serially reusable resources one at a time, request them one at a time and release one before requesting the next.
- Share resources as much as possible. If the requests in the lists shown in Figure 6-4 had been for shared control, there would have been no interlock. This does not mean you should share a resource that you will modify. It does mean that you should analyze your requirements for the resources carefully, and not request exclusive control when shared control is enough.
- Use the ENQ macro to request control of more than one resource at a time. The requesting program is placed in a wait condition until all of the requested resources are available. Those resources not being used by any other program immediately become exclusively available to the waiting program. For example, instead of coding the two ENQ macros shown in Figure 6-5, you could code the one ENQ macro shown in Figure 6-6. If all requests were made in this manner, the interlock shown in Figure 6-4 could not occur. All of the requests from one task would be processed before any of the requests from the second task. The DEQ macro can release a resource as soon as it is no longer needed; resources requested in a multiple ENQ can be individually released through separate DEQ instructions.

```
ENQ (NAME1ADD,NAME2ADD,E,8,SYSTEM)
ENQ (NAME3ADD,NAME4ADD,E,10,SYSTEM)
```

Figure 6-5. Two Requests For Two Resources

```
ENQ (NAME1ADD,NAME2ADD,E,8,SYSTEM,NAME3ADD,NAME4ADD,E,10,SYSTEM)
```

Figure 6-6. One Request For Two Resources

- If the use of one resource always depends on the use of a second resource, then you can define the pair of resources as one resource. On the ENQ and DEQ macros, define the pair with a single rname and qname. You can use this procedure for any number of resources that are always used in combination. However, the control program cannot then protect these resources if they are also requested independently. Any requests must always be for the set of resources.
- If there are many users of a group of resources and some of the users require control of a second resource while retaining control of the first resource, it is

still possible to avoid interlocks. In this case, each user should request control of the resources in the same order. For instance, if resources A, B, and C are required by many tasks, the requests should always be made in the order of A, B, and then C. An interlock situation will not develop, since requests for resource A will always precede requests for resource B.

Chapter 7. Program Interruption Services

The supervisor offers many services to detect and process abnormal conditions during system execution. The hardware detects certain types of abnormal conditions (such as an attempt to execute an instruction with an invalid operation code) and causes program interruptions to occur. The software detects other abnormal conditions (such as an attempt to open a data set that is not defined to the system, which causes the OPEN routine to request abnormal termination by issuing an ABEND macro).

Some conditions encountered in a program cause a program interruption. These conditions include incorrect parameters and parameter specifications, as well as exceptional results, and are known generally as program exceptions. You can disable the interruptions for certain exceptions (fixed point and decimal overflow, exponent underflow, and significance) by setting the corresponding bits in the program status word (PSW) to zero by means of the SPM instruction.

When a task becomes active for the first time, all program interruptions that can be disabled are disabled, and the task uses a standard system exit routine, included when the system was generated. This exit routine gets control when certain program interruptions occur; it issues an ABEND macro specifying task abnormal termination and requesting a dump.

Specifying User Exit Routines

By issuing the SPIE or ESPIE macro, you can specify your own exit routine to be given control for one or more types of program exceptions. If you issue an ESPIE macro, you can also pass the address of a parameter list to the exit routine. When one of the specified program exceptions occurs in a problem state program being executed in the performance of a task, the exit routine receives control in the key of the active task and in the addressing mode in effect when the SPIE or ESPIE was issued. (If a SPIE macro was issued, this is 24-bit addressing mode.) For other program interruptions, part of the system, the recovery termination manager (RTM), gets control. If the SPIE or ESPIE macro specifies an exception for which the interruption has been disabled, the system enables the interruption when the macro is issued.

If a program interruption occurs, the exit routine receives control on interrupt codes 0 through F. The interrupted program must be in primary mode, where the primary, home, and secondary address space is the same.

The environment established by an ESPIE macro exists for the entire task, until the environment is changed by another SPIE/ESPIE macro, or until the program creating the ESPIE returns. Each succeeding SPIE or ESPIE macro completely overrides specifications in the previous SPIE or ESPIE macro. You can intermix SPIE and ESPIE macros in one program. Only one SPIE or ESPIE environment is active at a time. If an exit routine issues an ESPIE macro, the new ESPIE environment does not take effect until the exit routine completes.

The system automatically deletes the SPIE/ESPIE exit routine when the request block (RB) that established the exit terminates. If a caller attempts to delete a specific SPIE/ESPIE environment established under a previous RB, the caller is abended with a system completion code of X'46D'. A caller can delete all previous

SPIE and ESPIE environments (regardless of the RB under which they were established) by specifying a token of zero with the RESET option of the ESPIE macro or an exit address of zero with the SPIE macro.

A program, executing in either 24-bit or 31-bit addressing mode in the performance of a task, can issue the ESPIE macro. If your program is executing in 31-bit addressing mode, you cannot issue the SPIE macro. The SPIE macro is restricted in use to callers executing in 24-bit addressing mode in the performance of a task. The following topics describe how to use the SPIE and ESPIE macros.

Using the SPIE Macro

The PICA and the program interruption element (PIE) contain the information that enables the system to intercept user-specified program interruptions established using the SPIE macro. You can modify the contents of the active PICA in order to change the active SPIE environment. The PICA and the PIE are described in the following topics.

Program Interruption Control Area

The expansion of each standard or list form of the SPIE macro contains a system parameter list called the program interruption control area (PICA). The PICA, as shown in Figure 7-1, contains the new program mask for the interruption types that can be disabled in the PSW, the address of the exit routine to be given control when one of the specified interruptions occurs, and a code for interruption types (exceptions) specified in the SPIE macro.

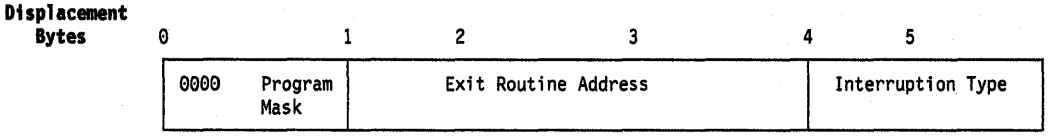


Figure 7-1. Program Interruption Control Area

The system maintains a pointer (in the PIE) to the PICA referred to by the last SPIE macro executed. This PICA might have been created by the last SPIE or might have been created previously and referred to by the last SPIE. Before returning control to the calling program or passing control to another program via an XCTL or XCTLX macro, each program that issues a SPIE macro must cause the system to adjust the SPIE environment to the condition that existed previously or to eliminate the SPIE environment if one did not exist on entry to the program. When you issue the standard or execute form of the SPIE macro to establish a new SPIE environment, the system returns the address of the previous PICA in register 1. If no SPIE/ESPIE environment existed when the program was entered, the system returns zeroes in register 1.

You can cancel the effect of the last SPIE macro by issuing a SPIE macro with no parameters. This action does not reestablish the effect of the previous SPIE; it does create a new PICA that contains zeroes, thus indicating that you do not want an exit routine to process interruptions. You can reestablish any previous SPIE environment, regardless of the number or type of subsequent SPIE macros issued, by using the execute form of the SPIE specifying the PICA address that the system returned in register 1. The PICA whose address you specify must still be valid (not overlaid). If you specify zeroes as the PICA address, the SPIE environment is eliminated.

Figure 7-2 shows how to restore a previous PICA. The first SPIE macro designates an exit routine called FIXUP that is to be given control if fixed-point overflow occurs.

The address returned in register 1 is stored in the fullword called HOLD. At the end of the program, the execute form of the SPIE macro is used to restore the previous PICA.

```
         .
         .
         SPIE   FIXUP,(8)   Provide exit routine for fixed-point overflow
         ST     1,HOLD      Save address returned in register 1
         .
         .
         L      5,HOLD      Reload returned address
         SPIE   MF=(E,(5))  Use execute form and old PICA address
         .
         .
HOLD     DC     F'0'
```

Figure 7-2. Using the SPIE Macro

Program Interruption Element

The first time you issue a SPIE macro during the performance of a task, the system creates a 32-byte program interruption element (PIE) in the virtual storage area assigned to your job step. Because the PIE is freed the first time you eliminate the SPIE environment (by specifying a PICA address of zero in the execute form of the SPIE macro or by specifying a SPIE with no parameters), the system also creates a PIE whenever you issue a SPIE macro and no PIE exists. The format of the PIE is shown in Figure 7-3.

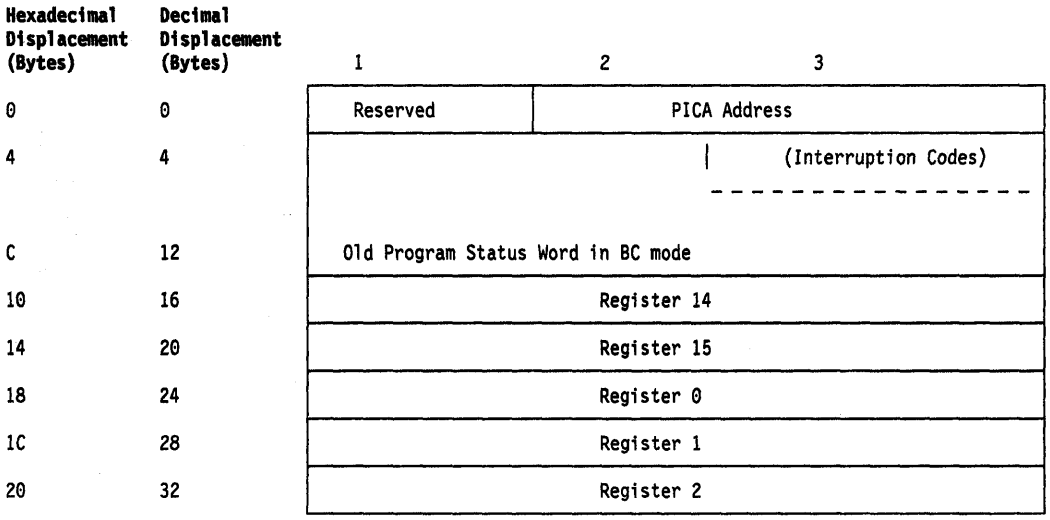


Figure 7-3. Program Interruption Element

The PICA address in the PIE is the address of the program interruption control area used in the last execution of the SPIE macro for the task. When control is passed to the routine indicated in the PICA, the BC mode old program status word contains the interruption code in bits 16-31 (the first byte is the exception extension code and the second is the exception code); you can test these bits to determine the cause of the program interruption. See *ESA/390 Principles of Operation* for an explanation of the format of the old program status word. The system stores the contents of registers 14, 15, 0, 1, and 2 at the time of the interruption as indicated.

Using the ESPIE Macro

The ESPIE macro extends the functions of the SPIE macro to callers in 31-bit addressing mode. The options that you can specify using the ESPIE macro are:

- SET to establish an ESPIE environment (that is, specify the interruptions for which the user-exit routine will receive control)
- RESET to delete the current ESPIE environment and restore the SPIE/ESPIE environment specified
- TEST to determine the active SPIE/ESPIE environment

If you specify ESPIE SET, you pass the following information to the service routine:

- A list of the program interruptions to be handled by the exit routine
- The location of the exit routine
- The location of a user-defined parameter list

The service routine returns a token representing the previously active SPIE or ESPIE environment, or zero if there was none.

If you code ESPIE RESET, you pass the token, which was returned when the ESPIE environment was established, back to the ESPIE service routine. The SPIE or ESPIE environment corresponding to the token is restored. If you pass a token of zero with RESET, all SPIE and ESPIE environments are deleted.

If you specify ESPIE TEST, you will be able to determine the active SPIE or ESPIE environment. An active SPIE environment is represented by a pointer to the PICA, which resides in user storage. (The PICA is described earlier in this section.) The active ESPIE environment is represented by protected control blocks belonging to the ESPIE service. To change an active ESPIE environment, you must issue the ESPIE macro with the SET or RESET option.

If an ESPIE environment is active and you issue a SPIE macro to specify interruptions for which a SPIE exit routine is to receive control, the service routine returns the address of a system-generated PICA in register 1. Do not modify the contents of the system-generated PICA; use them to restore the previous SPIE or ESPIE environment.

The Extended Program Interruption Element (EPIE)

The system creates an EPIE the first time you issue an ESPIE macro during the performance of a task or whenever you issue an ESPIE macro and no EPIE exists. The EPIE is freed when you eliminate the ESPIE environment.

The EPIE contains the information that the ESPIE service routine passes to the ESPIE exit routine when it receives control. When the exit routine receives control, register 1 contains the address of the EPIE. (See the topic "Register Contents Upon Entry to User's Exit Routine" for the contents of the other registers.) The format of the EPIE is shown in *Diagnosis: Data Areas*.

Environment Upon Entry to User's Exit Routine

When control is passed to your routine, the register contents are as follows:

Register 0: Internal control program information.

Register 1: Address of the PIE or EPIE for the task that caused the interruption.

Registers 2-12: Same as when the program interruption occurred.

Register 13: Address of the save area for the main program. The exit routine cannot use this area.

Register 14: Return address (to the system).

Register 15: Address of the exit routine. The exit routine must be in virtual storage when it is required, and must return control to the system using the address passed in register 14. For an ESPIE macro, the system restores all 16 registers from the EPIE. For a SPIE macro, the system restores registers 14, 15, 0, 1, and 2 from the program interruption element after control is returned, but does not restore the contents of registers 3-13. If a program interruption occurs when the program interruption exit routine is in control, the system exit routine gets control.

The access registers and linkage stack pointer have the values that were current at the time of the program interruption.

Functions Performed in User Exit Routines

Your exit recovery routine must determine the type of interruption that occurred before taking corrective action. Determining the type of interruption depends on whether the exit is associated with an ESPIE or a SPIE macro.

- For an ESPIE, your exit recovery routine can check the two-byte interruption code (the first byte is the exception extension code and the second is the exception code) at offset X'52' in the EPIE.
- For a SPIE, your exit recovery routine can test bits 16 through 31 (the first byte is the exception extension code and the second is the exception code) of the old program status word (OPSW in BC mode) in the PIE.

Note: For both ESPIE and SPIE — If you are using vector instructions and an exception of 8, 12, 13, 14, or 15 occurs, your recovery routine can check the exception extension code (the first byte of the two-byte interruption code in the EPIE or PIE) to determine whether the exception was a vector or scalar type of exception.

For more information about the exception extension code, see *IBM System/370 Vector Operations.*

Your recovery routine can alter the contents of the registers when control is returned to the interrupted program. The procedure for altering the registers also depends on whether the exit is associated with an ESPIE or a SPIE.

- For an ESPIE exit, the recovery routine can alter the contents of registers 0 through 15 in the save area in the EPIE because the system reloads these registers from this area when it returns control to the interrupted program.
- For a SPIE exit, the recovery routine can alter registers 14 through 2 in the register save area in the PIE because the system reloads these registers from this area when it returns control to the interrupted program. To change registers 3 through 13, the recovery routine must alter the contents of the registers.

The recovery routine can also alter the last four bytes of the OPSW in the PIE or EPIE. For an ESPIE, the recovery routine alters the CC and program mask starting at the third byte in the OPSW. By changing the OPSW, the routine can select any return point in the interrupted program. In addition, for ESPIE exits, the routine must set the AMODE bit of this four-byte address to indicate the addressing mode of the interrupted program.

Chapter 8. Program Termination and Dumping Services

The recovery termination manager (RTM) monitors the flow of control of software recovery processing and supplies the services of normal and abnormal task termination. RTM selects the appropriate recovery or termination process according to the status of the system.

You can write exit routines, called recovery routines, to handle certain types of interruptions and abnormal conditions. The supervisor initiates the recovery/termination process for your program either when you request it (for example, by issuing an ABEND macro) or when the system detects a condition that will degrade the system or destroy data.

One of the major purposes of a recovery routine is to find out what caused the program to terminate. Storage dumps that your recovery routines request through ABEND, SNAP, and SNAPX macros can provide you with information about the abending task.

Recovery/Termination Services

RTM gets control in response to events such as the following:

- Unanticipated program checks (except those protected by SPIE or ESPIE routines)
- Machine checks
- I/O error on page-in request
- Being in AR mode and issuing an SVC that is not supported in AR mode.
- Request by an authorized caller to terminate a task
- ABEND macros

RTM invokes any recovery routine that has been established to recover or clean up for the process in control. The recovery routine could be one of yours or it could be a system routine. If this recovery routine cannot recover from the incident (it requests termination or itself fails), RTM invokes the previously-established recovery routine. This passing of control from one recovery routine to another is called percolation. If none of the recovery routines can recover (request a retry), the system terminates the process in control.

The recovery routines you can establish are called **ESTAE-type** routines. You can establish an ESTAE-type routine in any of the following ways:

- ESTAE macro
- ESTAEX macro
- STAI parameter of the ATTACH macro
- ESTAI parameter of the ATTACH or ATTACHX macros

The ESTAEX macro and the ESTAE macro establish ESTAE-type recovery for your program. ESTAEX provides the same function as ESTAE, but also supports cross memory mode and access register ASC mode, and positions your code for future enhancements. IBM recommends that you issue ESTAEX rather than ESTAE.

The ESTAI parameter on the ATTACH or ATTACHX macros, and the STAI parameter on the ATTACH macro, establish recovery for a task and its subtasks. The STAI parameter is the pre-MVS/XA version of the ESTAI parameter.

You can also establish associated recovery routines (ARRs). ARRs are associated with PC routines. However, only authorized programs can create the cross memory environment required to use these routines. For information on coding ARRs, see the application development books that are available to the programmers in your installation that use authorized macros.

Recovery Routine Processing

When a recovery routine gets control, it determines why it has been entered and decides either to percolate or to retry. To tell RTM what it wants done, the recovery routine issues the SETRP macro, which manipulates fields in the system diagnostic work area (SDWA). When the recovery routine returns to RTM, RTM honors the request, if possible.

To allow communication between the main routine and the recovery routine, there is a parameter area. For a recovery routine established by an ESTAE macro, you can supply a parameter area by coding the PARAM parameter on the macro. When you establish a recovery routine, RTM saves a pointer to the parameter area and makes the pointer available to your recovery routine when it is entered. Usually, the main routine uses the parameter area to leave a footprint, that is, it sets indicators as part of normal processing; if an error occurs, these indicators let the recovery routine know where in the main process the failure occurred. The recovery routine can examine the footprint to determine what action to take.

If the recovery routine decides that a retry might be successful, it asks RTM to continue execution of the main routine at some appropriate point. Note that retry is not always allowed. If a recovery routine requests a retry when retry is not allowed, RTM ignores the request and continues with the termination process (percolates).

Any recovery routine that requests a retry must always include logic designed to avoid recursion, to prevent the creation of a tight loop between the recovery routine and the retry portion of the main routine. For example, if the recovery routine supplies a bad retry address to RTM, and the execution of the first instruction at the given address causes a program check, the first recovery routine to get control is the one that just requested the retry. If the recovery routine requests another retry at the same address, the loop is formed.

Using SETRP to Change the Completion and Reason Codes

You can specify both completion and reason code values on the ABEND macro. RTM passes these values to recovery exit routines to identify abnormal terminations. You can change the values of the completion code and the reason code by using the SETRP macro. The COMPCOD keyword allows you to specify a new completion code; the REASON keyword allows you to specify a new reason code.

The reason code has no meaning by itself, but must be used in conjunction with a completion code. In order to maintain meaningful completion and reason codes, RTM propagates changes to these values according to the following rules:

- If a user changes both the completion code and the reason code, RTM accepts both new values.
- If a user changes the reason code but not the completion code, RTM accepts the new reason code and uses the unchanged completion code.
- If a user changes the completion code but not the reason code, RTM accepts the new completion code and uses a zero for the reason code.
- If a user does not change either value, RTM uses the unchanged values.

Changing the Completion and Reason Codes Directly

Using the SETRP macro is the preferred way for changing the completion and reason codes. If you change these values directly in a recovery exit routine you should emulate SETRP processing as follows:

- When you change the completion code, store the new completion code in SDWACMPC, a three-byte field in the system diagnostic word area (SDWA), and set the one-bit flag, SDWACCF, to indicate the change.
- When you change the reason code, store the new reason code in SDWACRC, a four-byte field in the SDWA, and set the one-bit flag, SDWAREAF, to indicate the change.

Before passing control to a recovery exit routine, RTM saves the current completion and reason codes. After the recovery routine returns control to RTM, RTM examines the contents of the SDWACCF and SDWAREAF flags to determine whether changes have been made to the completion and reason codes and then determines which values to pass to the next recovery exit routine. RTM makes this decision as shown in the following table:

SDWACCF Completion code flag	SDWAREAF Reason code flag	Values passed to the next recovery exit routine
ON	OFF	The abend completion code and a reason code of zero
OFF	ON	The unchanged completion code and the altered reason code
ON	ON	The altered completion code and the altered reason code

If both flags are off, RTM passes the values in the user's SDWA to the next recovery exit routine.

Handling Abnormal Conditions

The system does a great deal of checking for abnormal conditions. It uses hardware to detect errors such as protection violations or addressing errors. The data management and supervisor routines provide some error checking facilities to ensure that, based on the information you have provided, only valid data is being processed, and that you have not made any conflicting requests. For abnormal conditions that can possibly be corrected, the system returns to your program with a return code indicating the probable source of the error. For conditions that indicate that further processing would result in degradation of the system or destruction of data, the system gives control to RTM.

There will, of course, be abnormal conditions unique to your program that the system cannot detect. Figure 8-1 is an example of one of these. The routine shown in Figure 8-1 checks a control field in an input parameter list to determine which function the program is to perform. Only characters 1 through 4 are valid in the control field. The presence of any other character is invalid, but the routine must be prepared to detect and handle these characters. One way to handle an invalid character is to return to the calling program with an error return code. The calling program can then try to interpret the return code and recover from the error. If it cannot do so, the calling program can detach its incomplete subtasks, execute its usual termination procedures, and return control to its calling program, again with an error return code. This procedure might result in termination of all the tasks of a job step; if it does, you can use the COND parameters of the JOB and EXEC statements to indicate whether subsequent job steps should be executed.

Another way to handle this unexpected condition is to issue an ABEND macro to give RTM control.

The position within the job step hierarchy of the task for which the ABEND macro is issued determines the exact function of the abnormal termination routine. If an ABEND macro is issued when the job step task (the highest level or only task) is active, or if the STEP parameter is coded in an ABEND macro issued during the performance of any task in the job step, all the tasks in the job step are terminated. For example, if the STEP parameter is coded in an ABEND macro under TSO, the TSO job will be terminated. An ABEND macro (without a STEP parameter) that is issued in performance of any task in the job step task usually causes only that task and its subtasks to be abnormally terminated. However, if the abnormal termination cannot be fulfilled as requested, it might be necessary for RTM to abnormally terminate the job step task.

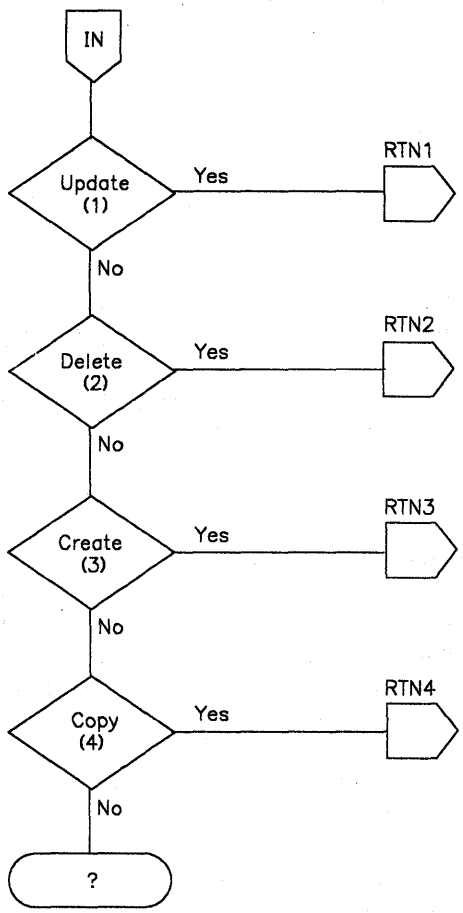


Figure 8-1. Detecting an Abnormal Condition

If you have created a recovery routine for your program, RTM passes control to your routine. If you have not set up a recovery routine, RTM handles the problem. The action RTM takes depends on whether or not the job step is going to be terminated.

If the job step is not going to be terminated, RTM:

- Releases the resources owned by the terminating task and all of its subtasks starting with the lowest level task.
- Places the system or user completion code specified in the ABEND macro in the task control block of the active task (the task for which the ABEND macro was issued).
- Posts the ECB with the completion code specified in the ABEND macro if the ECB parameter was coded in the ATTACH or ATTACHX macro issued to create the active task.
- Schedules the end-of-task exit routine to be given control when the originating task becomes active if the ETXR parameter was coded in the ATTACH or ATTACHX macro issued to create the active task.
- If neither the ECB nor ETXR parameter were specified by the ATTACH or ATTACHX macro, RTM calls a routine to FREEMAIN the terminating TCB.

If the job step is to be terminated, RTM:

- Releases the resources owned by each task, starting with the lowest level task, for all tasks in the job step. No end-of-task exit routine is given control.
- Writes the system or user completion code specified in the ABEND macro on the system output device.

The remaining steps in the job are skipped unless you can establish your own recovery routine to perform similar functions and any other functions that your program requires. Use either the ESTAE macro or the ATTACH or ATTACHX macro with the ESTAI option to set up a recovery routine that gets control whenever your program issues an ABEND macro. If your program is running in AR address space control (ASC) mode, use the ESTAEX or ATTACHX macro. The recovery routine that gets control will have the ASC mode of the caller.

Your recovery routine also gets control if the system issues an ABEND on your behalf. Your routine can determine its actions with regard to the abnormal condition. With this approach, you can put less error handling code in your mainline routines. For example, there is no need to check return codes after a subroutine if the subroutine issues an ABEND. The error handling functions can be part of the ESTAE or ESTAI routines that execute only when there is an error.

Whether you use ESTAE or ESTAEX to establish an ESTAE-type recovery routine depends on the ASC mode of your program and whether it is in cross memory mode. To issue an ESTAE macro, your program must be in primary mode, where primary and home address spaces are the same. (In other words, your program cannot be in cross memory mode.) The ESTAE service routine is entered through an SVC. To issue the ESTAEX macro, your program can be in either primary or AR mode and can also be in cross memory mode.

How to Use an ESTAE-type Recovery Routine

Within an ESTAE recovery routine, you can perform pre-termination functions and diagnose the error. You can also determine whether abnormal termination should continue for the task, or whether normal processing can continue at some point in the mainline routine.

When the abnormal termination is issued, the ESTAE recovery routine must be resident. It can either be part of the program issuing the ESTAE or be brought into virtual storage with the LOAD macro before the ESTAE-type routine is established.

A single program can create more than one recovery routine by issuing the ESTAE macro with the CT parameter. (The program can also overlay or delete recovery routines by issuing ESTAE macros with the OV parameter or with an address of zero, respectively.) All ESTAE requests issued by programs running under the same task are queued so that the routine established by the most recent ESTAE request is the first to get control. If this routine fails or requests that abnormal termination continue (percolation), RTM deletes the routine, and the exit established by the previous ESTAE request gets control.

If you want to use the same recovery routine for several tasks at the same time, the routine must be reenterable. For convenience, you should make all your ESTAE exit routines reenterable.

You must delete all the ESTAE routines you have created before returning control to your caller. If you try to delete an ESTAE routine not associated with your request block (RB), you get a return code that indicates your request is invalid.

Your ESTAE-type recovery routine might provide information that dump analysis and elimination (DAE) can use to construct unique symptom strings needed to describe software failures. DAE uses these symptom strings to analyze dumps and suppress duplicates as requested. Each symptom string contains specific pieces of information called symptoms that DAE obtains from fields in the system diagnostic work area (SDWA), SDWA extensions, ABDUMP symptom area, and SDWA variable recording area (SDWAVRA).

When using DAE, you must select symptoms carefully. If the data you supply is too precise, no other failure will have the same symptoms; if the data is too general, many failures will have the same symptoms. For information on DAE, see *Planning: Problem Determination and Recovery*.

Interface to an ESTAE-type Recovery Routine

Before your first ESTAE-type routine receives control, RTM can handle outstanding I/Os at the time of the failure. You request this through the macro that established the routine (that is, through the PURGE parameter on ESTAE or ATTACH). RTM performs the requested I/O processing only for the first ESTAE-type routine. Subsequent routines receive an indication of the I/O processing previously done, but no additional processing is performed. For ARRs, you do not have a choice; I/O processing continues normally.

During processing of the first and all subsequent recovery routines, RTM allows or disallows asynchronous processing depending on how you specify the ASYNCH parameter when you establish the routine (that is, through the ASYNCH parameter on ESTAE and ATTACH.)

The recovery routine is enabled and has the same protection key and PSW key mask (PKM) as the routine that established the recovery routine as long as the establishing routine was under a problem program protection key (keys 8-15). A routine created by a program running under key 0-7 gets control in key 0.

Before each ESTAE-type recovery routine receives control, RTM tries to get storage for and to initialize a work area to contain information about the error. This work area is called the system diagnostic work area (SDWA). To access the SDWA, you must include the SDWA mapping macro - IHASDWA - as a DSECT in your routine. The the SDWA fields are as follows:

Field Name	Use
SDWAPARM	For ESTAE-type routines, this four-byte field, contains the pointer to the user parameter list that you supply for an ESTAE-type recovery routine. For routines established through ESTAEX, this field contains the pointer to an eight-byte area. The eight-byte area points to the user parameter list that you supply for a recovery routine; the first four bytes contain the address of the parameter list and the second four bytes contain the ALET that identifies the address space or data space.
SDWACMPC	This three-byte field contains the ABEND completion code that existed when RTM gave control to the recovery routine. The recovery routine can change the ABEND code by changing this field. The system code appears in the first twelve bits and the user code appears in the second twelve bits.
SDWAGRSV	This field shows the contents of general registers 0-15 as they were at the time of the error.
SDWAARER	This field contains access registers 0-15 as they were at the time of the error.
SDWACRC	This four-byte field contains the reason code that existed when RTM entered the recovery routine. The recovery routine can change the reason code by changing this field.
SDWAEC1	This field contains the PSW that existed at the time of the error.
SDWAEC2	The contents of this field vary according to the type of recovery routine: • For an ESTAE-type routine, the field contains the extended control PSW of the RB that created the recovery routine at the time the RB last incurred an interruption. • For an ESTAI routine, this field contains zeroes.
SDWASRSV	The contents of this field vary according to the type of recovery routine. • For an ESTAE-type routine, this field contains the general registers 0-15 as they were when the RB that established the recovery routine incurred an interruption, or a BAKR or PC instruction placed an entry on the linkage stack. • For an ESTAI routine, this field contains zeroes. If the recovery routine requests a retry, RTM uses the contents of this field to load the registers for the retry routine. To change the contents of the registers for the retry routine, you must make the changes to this field and request a register update (RETREGS=YES) on the SETRP macro.
SDWAARSV	The contents of this field vary according to the type of recovery routine: • For an ESTAE-type routine, this field contains the access registers 0 - 15 as they were when the RB that established the recovery routine incurred an interruption, or a BAKR or PC instruction placed an entry on the linkage stack. • For an ESTAI routine, this field contains zeroes. If the recovery routine requests a retry, RTM uses the contents of this field to load the access registers for the retry routine. To change the contents of the registers for the retry routine, you must make the changes in this field, then request a register update (RETREGS=YES) on the SETRP macro.
SDWASPID	This field contains the subpool ID of the SDWA.
SDWALNTH	This field contains the length, in bytes, of the SDWA.
SDWACOMU	The recovery routines use this 8-byte field to communicate with each other when percolation occurs. RTM copies this field from one SDWA to the next on all percolations. If the field contains zeroes, either there was no information passed or RTM was not able to pass it.
SDWAVRAL	This field contains the length of the variable recording area (VRA) for this SDWA.

Field Name	Use
SDWAHEX	This is a one bit field set by the recovery routine to indicate that EREP is to print the data in the VRA in hexadecimal form.
SDWAEBC	This is a one-bit field set by the recovery routine to indicate that EREP is to print the data in the VRA in EBCDIC form.
SDWAURAL	This is a one-byte field set by the recovery routine to indicate the length of the VRA used. The field initially contains zeroes. Whenever the recovery routine uses any part of the VRA, it must set this field.
SDWACCF	The recovery routine sets this one-bit field when it changes the completion code.
SDWAREAF	The recovery routine sets this one-bit field when it changes the reason code.
SDWAFAIN	This 12-byte field contains the six bytes of the instruction stream that both precede and follow the failing instruction pointed to by the PSW. The SDWAFAIN field contains zeroes if RTM cannot access the failing instruction stream pointed to by the time-of-error PSW. For example, if the time-of-error PSW is not valid, the SDWAFAIN field contains zeroes.
SDWADAET	This eight-byte field contains DAE status and error flags for this dump.
SDWAOCUR	This two-byte field contains the number of previous occurrences of these symptoms in other SDWAs.
SDWATRAN	This field contains the translation exception address, if a translation exception occurred.
SDWATEAR	For callers in AR mode, this field identifies the access register that the program was using when the translation exception occurred.

The register contents on entry to the ESTAE-type routine depends on whether or not RTM obtained an SDWA. If RTM obtained an SDWA, the registers on entry to the recovery routine are as follows:

Register 0 A code indicating the type of I/O processing performed:

- 0 — Active I/O has been quiesced and is restorable.
- 4 — Active I/O has been halted and is not restorable.
- 8 — No I/O was active when the ABEND occurred.
- 16 — No I/O processing was performed.

Register 1 Address of the SDWA.
Register 13 Address of a 72-byte register save area.
Register 14 Return address.
Register 15 Entry point address.

All other registers are used as work areas by the system.

The SDWA resides in the primary address space and the access registers are set to zeroes.

When the ESTAE-type routine has completed its analysis of the error, it can use the SETRP macro to inform RTM what it wants done. The SETRP macro initializes the SDWA with the desired options. You can return from the ESTAE exit routine by using the SETRP REGS parameter or by using a BR 14 instruction.

If RTM could not obtain an SDWA, the general purpose register contents on entry to the recovery routine are as follows:

Register 0	12 (decimal). RTM could not obtain an SDWA.
Register 1	ABEND completion code.
Register 2	Address of the user-supplied parameter list if you issued the ESTAE macro or the ESTAEX macro. Otherwise, 0. For callers in AR mode, AR 2 contains an ALET that qualifies the address in GPR 2 if you issued the ESTAEX macro; otherwise, AR 2 contains 0.
Register 14	Return address.
Register 15	Entry point address.

All other registers are used as work areas by the system.

If RTM could not provide an SDWA, it does not provide a register save area either. In this case, your ESTAE-type routine must save the address in register 14 and use it as the return address to RTM. You must place a return code in register 15 before returning to RTM. The return code indicates whether ABEND processing is to be continued for the task or whether a retry address can be given control. The return codes are:

Return Code	Meaning
0	Continue with termination. Any ESTAE-type routines that were established prior to this routine will get control.
4	Give control to the retry address. (You must place the retry address in register 0.)

How to Use an ESTAI Routine

You can provide an exit in your program to intercept abnormal termination of a subtask by using the ESTAI parameter on the ATTACH or ATTACHX macro you issue to create the subtask. Once you establish an ESTAI routine for one subtask, that routine will be used for all of the subtask's subtasks. For example, suppose task A attaches task B and uses the ESTAI parameter on ATTACH or ATTACHX. When task B attaches task C, the ESTAI routine created by task A is active for C as well as B.

The ESTAI routine receives control under the failing task's TCB.

Because more then one subtask can abnormally terminate at the same time, the ESTAI routine might be used by more than one subtask concurrently. Your ESTAI exit routines must therefore be reenterable.

Interface to an ESTAI Routine

ESTAI routines are entered after all ESTAE routines that exist for a given task have received control and have either failed or percolated. The interface to ESTAI routines is the same as for ESTAE exits, however, one additional option is available for ESTAI. When you return to RTM, you can specify return code 16 either on the SETRP macro if an SDWA exists, or in register 15 if an SDWA is not available. The return code indicates to RTM that termination should continue and that no other ESTAI routines should receive control for that task.

ESTAE-type Retry Routines

If an ESTAE-type routine requests percolation, RTM gives control to the next oldest ESTAE or ESTAI routine that exists for the task. However, if a given ESTAE or ESTAI exit routine requests retry, the system takes a dump if requested and transfers control to the retry routine without processing any further ESTAE-type routines.

An ESTAE or ESTAI routine can request retry whenever the SDWACLUP bit in the SDWA is set to zero. To request retry, the exit routine must supply a retry address. The retry address is the point in the mainline routine that is to get control in order to continue its processing. In response to a valid retry request, RTM gives control to the retry address supplied. A retry routine requested by an ESTAE-type routine operates as an extension of the mainline code; it operates under the same RB and in the same addressing mode as the caller that established the ESTAE-type routine. The system purges all RBs in the chaining order created after the retry RB before it gives control to the retry routine.

RTM purges the RB queue to cancel the effects of partially executed programs that are at a lower level in the program hierarchy than the program for which the retry occurs. Certain effects, however, cannot be canceled. Among these are:

- Subtasks created by an RB to be purged
- Resources allocated by the ENQ macro
- DCBs that exist in dynamically-acquired virtual storage

If there are quiesced restorable I/O operations, the retry routine can restore them. RTM supplies a pointer to the purged I/O restore list in register 2. You can use the RESTORE macro to have the system restore all I/O requests on the list. The retry routine should free the storage occupied by the SDWA (if there was an SDWA) when that storage is no longer needed unless the exit routine specified FRESDWA=YES on the SETRP macro. The subpool number and length to use on the FREEMAIN macro are in the SDWA.

Interface to a Retry Routine

There are two different interfaces to a retry routine:

- If RTM was able to obtain an SDWA, you can set the register contents in the SDWA to whatever you wish and request that they be passed to the retry routine by coding RETREGS=YES in the SETRP macro. This method is used most often in mainline processing.
- If RTM could not obtain an SDWA or if RETREGS=NO was specified on the SETRP macro, only parameter registers are passed to the retry routine. This method is used most often if a special retry routine is to get control.

If RTM could not obtain an SDWA, the contents of the relevant registers are as follows:

Register 0	12 (decimal)
Register 1	Address of the user parameter list if established through the ESTAE, ATTACH, or ATTACHX macro; address of a doubleword containing address and ALET of parameter list if established through ESTAEX macro
Register 2	Address of the purged I/O restore list if I/O was quiesced and is restorable; otherwise, 0
Register 14	Return address
Register 15	Entry point address of the retry routine

If RTM obtained an SDWA and the retry routine specified RETREGS=NO and FRESDWA=NO, the contents of the relevant registers are as follows:

Register 0	0
Register 1	Address of the SDWA
Register 2	Address of the purged I/O restore list if I/O was quiesced and is restorable; otherwise, 0
Register 14	Return address
Register 15	Entry point address of the retry routine

If RTM obtained an SDWA and the retry routine specified RETREGS=NO and FRESDWA=YES, the contents of the relevant registers are as follows:

Register 0	20 (decimal)
Register 1	Address of the user parameter list if established through the ESTAE, ATTACH, or ATTACHX macro; address of a doubleword containing address and ALET of parameter list if established through ESTAEX macro
Register 2	Address of the purged I/O restore list if I/O was quiesced and is restorable; otherwise, 0
Register 14	Return address
Register 15	Entry point address of the retry routine

If the retry routine requested register update (RETREGS=YES) before passing control to the retry routine, RTM restores the general purpose registers from SDWASRSV and the access registers from SDWAARSV. In this case, the recovery routine provides the contents of the registers for the retry routine by updating any or all of the register slots in the SDWASRSV before returning control to RTM with the retry request. If the recovery routine does not also request that the SDWA be freed, it must keep a pointer to the SDWA; this pointer enables the retry routine to reference and subsequently free the SDWA. Note that register 15 does not contain the entry point address of the retry routine unless the recovery routine sets it up that way.

In all cases, the routine runs enabled and the protection key is the same key as the routine that established the ESTAE-type recovery routine.

Summary of Recovery Routine Characteristics

Figure 8-2 summarizes the environment of the caller for ESTAE-type routines at three different times:

- At the time of issuing the macro
- At the time of the entry to the recovery routine
- At the time of entry to the retry routine

Note a condition under which the last two columns might not be correct. If a caller of ESTAE issues a stacking PC or BAKR instruction in the recovery routine and percolation occurs, the current stack entry at time of entry to the percolated-to ESTAE will be more recent than the entry at the time of the original error.

Macro Service	Environment		
	When macro was issued	**At entry to recovery routine**	**At entry to retry routine**
STAE or ESTAE	ASC mode = primary PASN = SASN = HASN	ASC mode = primary PASN = SASN = HASN Linkage stack at time macro was issued	ASC mode = primary PASN = SASN = HASN Linkage stack at time macro was issued
ESTAEX	ASC mode = primary or AR	ASC mode at time macro was issued PASN and SASN at time macro was issued Linkage stack at time of error	ASC mode at time macro was issued PASN and SASN at time macro was issued Linkage stack at time macro was issued
ESTAI (through ATTACH or ATTACHX)	ASC mode = primary or AR PASN = SASN = HASN	ASC mode at time macro was issued PASN = SASN = HASN Linkage stack at time of error	ASC mode at time macro was issued PASN = SASN = HASN Linkage stack at time subtask was created
STAI (through ATTACH or ATTACHX)	ASC mode = primary PASN = SASN = HASN	ASC mode = primary PASN = SASN = HASN Linkage stack at time of error	ASC mode = primary PASN = SASN = HASN Linkage stack at time subtask was created
There is no restriction on AMODE at time of invocation for any of the above services. At time of entry to the recovery routine, the AMODE will be the same as the time of invocation, except for routines established through the ESTAEX macro. These routines always receive control in AMODE 31. The AMODE at the retry point will be the same as the AMODE on entry to the recovery routine.			

Figure 8-2. Summary of the Environments of Recovery Routines

Dumping Services

A problem program can request two types of storage dumps:

- An ABEND dump obtained through use of the DUMP parameter in the ABEND macro or the DUMP=YES parameter on the SETRP macro in a recovery exit.
- A snap dump obtained through use of the SNAP macro.

ABEND Dumps

An ABEND macro initiates error processing for a task. The DUMP option of ABEND requests a dump of storage and the DUMPOPT or DUMPOPX option may be used to specify the areas to be displayed. These dump options may be expanded by an ESTAE or ESTAI routine. The system usually requests a dump for you when it issues an ABEND macro. However, the system can provide an ABEND dump only if you include a DD statement (SYSABEND, SYSMDUMP, or SYSUDUMP) in the job step. The DD statement determines the type of dump provided and the system dump options that are used. When the dump is taken, the dump options that you requested (specified in the ABEND macro or by recovery routines) are added to the installation-selected options.

When writing an ESTAE-type recovery routine, note that the system accumulates the SYSABEND/SYSUDUMP/SYSMDUMP dump options specified by means of the SETRP macro and places them in the SDWA. During percolation, these options are merged with any dump options specified on an ABEND or CALLRTM macro or by other recovery routines. Also, the CHNGDUMP operator command can add to or override the options. The system takes one dump as specified by the accumulated options. If the recovery routine requests a retry, the system processes the dump before the retry. If the recovery routine does not request a retry, the system percolates through all recovery routines before processing the dump.

Obtaining a Symptom Dump

With all ABEND dumps, you will automatically receive a short symptom dump of approximately ten lines. This symptom dump provides a summary of error information, which will help you to identify duplicate problems.

You will receive this dump even without a DD statement unless your installation changes the default via the CHNGDUMP operator command or the dump parmlib member for SYSUDUMP.

SNAP Dumps

A program can request a SNAP dump at any time during its processing by issuing a SNAP macro. For a SNAP dump, the DD statement can have any name except SYSABEND, SYSMDUMP, and SYSUDUMP.

If your program is in AR ASC mode, use the SNAPX macro instead of SNAP. Make sure that the SYSSTATE ASCENV=AR macro has been issued to tell the macro to generate code and addresses appropriate for callers in AR mode.

Like the ABEND dump, the data set containing the dump can reside on any device that is supported by BSAM. The dump is placed in the data set described by the DD statement you provide. If you select a printer, the dump is printed immediately. However, if you select a direct access or tape device, you must schedule a separate job to obtain a listing of the dump, and to release the space on the device.

To obtain a dump using the SNAP macro, you must provide a data control block and issue an OPEN macro for the data set before issuing any SNAP macros. If the standard dump format is requested, 120 characters per line are printed. The data control block must contain the following parameters: DSORG=PS, RECFM=VBA, MACRF=W, BLKSIZE=882 or 1632, and LRECL=125. (The data control block is described in *Managing Non-VSAM Data Sets*, and *Macro Instructions for Non-VSAM Data Sets*). If a high-density dump is to be printed on a 3800 Printing Subsystem, 204 characters per line are printed. To obtain a high-density dump, code CHARS=DUMP on the DD statement describing the dump data set. The BLKSIZE= must be either 1470 or 2724, and the LRECL= must be 209. CHARS=DUMP can also be coded on the DD statement describing a dump data set that will not be printed immediately. If CHARS=DUMP is specified and the output device is not a 3800, print lines are truncated and print data is lost. If your program is to be processed by the loader, you should also issue a CLOSE macro for the SNAP data control block.

Finding Information in a SNAP Dump

You will obtain a dump index with each SNAP dump. The index will help you find information in the dump more quickly. Included in the information in the dump index is an alphabetical list of the active load modules in the dump along with the page number in the dump where each starts.

Obtaining a Summary Dump

You can request a summary dump for an abending task by coding the SUM option of the SNAP macro. You can also obtain a summary dump by coding the DUMPOPT option on the ABEND or SETRP macro and specifying a list form of SNAP that contains the SUM option. Use the DUMPOPX parameter on ABEND or SETRP to obtain an ABEND dump that contains data space storage. When you use DUMPOPX, specify a list form of SNAPX that contains the SUM option.

If SUM is the only option that you specify, the dump will contain a dump header, control blocks, and certain other areas. The contents of the summary dump are described in *Planning: Problem Determination and Recovery*.

Reporting Symptom Records

An installation's programmers can write authorized or unauthorized applications that detect and collect information about errors in their processing. Through the SYMRBLD or SYMREC macro, the applications can write a symptom record for each error out to SYS1.LOGREC, the online repository where MVS collects error information. Programmers can analyze the records in SYS1.LOGREC to diagnose and debug problems in the installation's applications.

The unit of information stored in SYS1.LOGREC is called a **symptom record**. The data in the symptom record is a description of some programming failure combined with a description of the environment where the failure occurred. Some of the information in the symptom record is in a special format called the **SDB** (structured data base) format.

In summary, an installation's programmers can:

- Build the symptom records using the SYMRBLD macro.
- Record the symptom records on SYS1.LOGREC using SYMRBLD or SYMREC.
- Format the symptom records into various kinds of reports using EREP and IPCS.

Writing Symptom Records to SYS1.LOGREC

Your application can build and write symptom records to SYS1.LOGREC one of two ways:

- Through invoking the SYMRBLD macro services
- By filling in fields of the ADSR mapping macro, then invoking SYMREC.

SYMRBLD services use both the ADSR mapping macro and SYMREC, thus decreasing the amount of code your application requires to write symptom records. IBM recommends that you use SYMRBLD rather than coding your application to directly use ADSR and SYMREC.

By invoking the SYMRBLD macro multiple times, you can generate code to build the symptom record. After storing all symptoms into the symptom record by using the SYMRBLD macro, invoke the SYMRBLD macro with the INVOKE=YES parameter one last time to write the data from the symptom record to SYS1.LOGREC.

The Format of the Symptom Record

The symptom record consists of six sections that are structured according to the format of the ADSR DSECT. These sections are numbered 1 through 5, including an additional section that is numbered 2.1. Because sections 2.1, 3, 4, and 5 of the symptom record are non-fixed, they do not need to be sequentially ordered within the record. In section 2, the application supplies the offset and the length of the non-fixed sections. The ADSR format is described in the *Diagnosis: Data Areas*, and the purpose of each section is as follows:

Section 1 (Environmental data): Section 1 contains the record header with basic environmental data. The environmental data provides a system context within which the software errors can be viewed. The SYMREC caller initializes this area to zero and stores the characters "SR" into the record header. The environmental data of section 1 is filled in automatically when you invoke the SYMREC macro. Section 1 includes items such as:

- CPU model and serial number
- Date and time, with a time zone conversion factor
- Customer assigned system name
- Product ID of the control program

Section 2 (Control data): Section 2 contains control information with the lengths and offsets of the sections that follow. The application must initialize the control information before invoking SYMREC for the first time. You can initialize the control information by using SYMRBLD with the INITIAL parameter. Section 2 immediately follows section 1 in the symptom record structure.

Section 2.1 (Component data): Section 2.1 contains the name of the component in which the error occurred, as well as other specific component-related data. The application must also initialize section 2.1 before invoking SYMREC. You can initialize the control information by using SYMRBLD with the INITIAL parameter.

Section 3 (Primary SDB symptoms): Section 3 contains the primary string of problem symptoms, which may be used to perform tasks such as duplicate problem recognition. When an application detects an error, it must store a string of symptoms in section 3, and this string becomes the primary symptom for the error. This string should be a unique and complete description of the error. All incidences of that error should produce the same string in section 3. When problems are analyzed, problems that have identical strings in section 3 represent the same error. Note that an application does not store any primary symptom string or invoke SYMREC unless it detects an error in its own processing. You can invoke SYMRBLD with the PRIMARY parameter to store symptoms into section 3.

Section 4 (Secondary SDB symptoms): Section 4 contains an optional **secondary symptom string**. The purpose of the secondary string is to provide additional symptoms that might supplement the symptoms in section 3.

Section 5 (Free-format data): Section 5 contains logical segments of optional problem-related data to aid in problem diagnosis. However, the data in section 5 is not in the SDB format, which is found in only sections 3 and 4. Each logical segment in section 5 is structured in a **key-length-data** format.

Symptom Strings — SDB Format

The symptom strings placed in sections 3 and 4 of the symptom record must be in the SDB (structured data base) format. In this format, the individual symptoms in sections 3 and 4 consist of a prefix and its associated data. For more information on the prefixes that SYMRBLD or SYMREC recognize, see *Problem Determination Guide*. Examples of typical prefixes are:

Prefix	**Data**
PIDS/	a component name
RIDS/	a routine name
AB/	an abend code
PRCS/	a return code

For a full explanation of symptom strings and how to format and print the four basic symptom record reports, see *Problem Determination Guide* and *Diagnosis: Using Dumps and Traces*.

Programming Notes for SYMREC Applications

This section contains programming notes on how the various fields of the ADSR data area (symptom record) are set. Some fields of the ADSR data area (symptom record) must be set by the caller of the SYMREC macro, and other fields are set by the system when the application invokes the SYMREC service. The fields that the SYMREC caller must always set are indicated by an RC code in the following sections. The fields that are set by SYMREC are indicated by an RS code.

The RA code designates certain flag fields that need to be set only when certain kinds of alterations and substitutions are made in the symptom record after the incident occurs. These alterations and substitutions must be obvious to the user who interprets the data. Setting these flag fields is the responsibility of the program that alters or substitutes the data. If a program changes a symptom record that is already written on the repository, it should set the appropriate RA-designated flag-bit fields as an indication of how the record has been altered.

The remaining fields, those not marked by RC, RS, or RA, are optionally set by the caller of the SYMREC macro. When SYMREC is invoked, it checks that all the

required input fields in the symptom record are set by the caller. If the required input fields are not set, SYMREC issues appropriate return and reason codes.

Programming Notes for Section 1

Notes in this section pertain to the following fields, which are in section 1 of the ADSR data area.

ADSRID	Record header	(RC)
ADSRGMT	Local Time Conversion Factor	
ADSRTIME	Time stamp	(RS)
ADSRTOD	Time stamp (HHMMSSTH)	
ADSRDATE	Date (YYMMDD)	
ADSRSID	Customer Assigned System/Node Name	(RS)
ADSRSYS	Product ID of Base System (BCP)	(RS)
ADSRCML	Feature and level of Symrec Service	(RS)
ADSRTRNC	Truncated flag	(RS)
ADSRPMOD	Section 3 modified flag	(RA)
ADSRSGEN	Surrogate record flag	(RA)
ADSRSMOD	Section 4 modified flag	
ADSRNOTD	ADSRTOD & ADSRDATE not computed flag	(RS)
ADSRASYN	Asynchronous event flag	(RA)
ADSRDTP	Name of dump	

Notes:

1. SYMREC stores the TOD clock value into ADSRTIME when the incident occurs. However, it does not compute ADSRTOD and ADSRDATE when the incident occurs, but afterwards, when it formats the output. When the incident occurs, SYMREC also sets ADSRNOTD to 1 as an indication that ADSRTOD and ADSRDATE have not been computed.
2. SYMREC stores the customer-assigned system node name into ADSRSID.
3. SYMREC stores the first four digits of the base control program component id into ADSRSYS. The digits are 5752, 5759 and 5745 respectively for MVS, VM*, and DOS/VSE*
4. The ADSRDTP field is not currently used by the system.
5. If some application creates the record asynchronously, that application should set ADSRASYN to 1. 1 means that the data is derived from sources outside the normal execution environment, such as human analysis or some type of machine post-processing.
6. If SYMREC truncates the symptom record, it sets ADSRTRNC to 1. This can happen when the size of the symptom record provided by the invoking application exceeds SYMREC's limit.
7. ADSRSGEN indicates that the symptom record was not provided as 'first time data capture' by the invoking application. Another program created the symptom record. For instance, the system might have abended the program, and created a symptom record for it because the failing program never regained control. Setting the field to 1 means that another program surrogate created the record. The identification of the surrogate might be included with other optional information, for example, in section 5.

---

* VM is a trademark of the IBM Corporation.

* VSE is a trademark of the IBM Corporation.

8. The application invoking SYMREC must provide the space for the entire symptom record, and initialize that space to hex zeroes. The application must also store the value 'SR' into ADSRID.
9. The fields ADSRCPM through ADSRFL2, which appear in the record that is written to SYS1.LOGREC, are also written back into the input symptom record as part of the execution of SYMREC.

Programming Notes for Section 2

Notes in this section pertain to the following fields, which are in section 2 of the ADSR data area.

```
ADSRARID   Architectural level designation        (RS)
ADSRL      Length of section 2                    (RC)
ADSRCSL    Length of section 2.1                  (RC)
ADSRCSO    Offset of section 2.1                  (RC)
ADSRDBL    Length of section 3                    (RC)
ADSRDBO    Offset of section 3                    (RC)
ADSRROSL   Length of section 4
ADSRROSA   Offset of section 4
ADSRRONL   Length of section 5
ADSRRONA   Offset of section 5
ADSRRISL   Length of section 6
ADSRRISA   Offset of section 6
ADSRSRES   Reserved for system use
```

Notes:

1. The invoking application must ensure that the actual lengths of supplied data agree with the lengths indicated in the ADSR data area. The application should not assume that the SYMREC service validates these lengths and offsets.
2. The lengths and offsets in section 2 are intended to make the indicated portions of the record indirectly addressable. Invoking applications should not program to a computed absolute offset, which may be observed from the byte assignments in the data area.
3. The value of the ADSRARID field is the architectural level at which the SYMREC service is operating. This field is supplied by the SYMREC service.
4. Section 2 has a fixed length of 48 bytes. Optional fields (not marked with RC, RS, or RA) will contain zeroes when the invoking application provides no values for them.

Programming Notes for Section 2.1

Notes in this section pertain to the following fields, which are in section 2.1 of the ADSR data area.

ADSRC	C'SR21' Section 2.1 Identifier	(RC)
ADSRCRL	Architectural Level of Record	(RC)
ADSRCID	Component identifier	
ADSRFLC	Component Status Flags	
ADSRFLC1	Non-IBM program flag	(RC)
ADSRLVL	Component Release Level	(RC)
ADSRPTF	Service Level	
ADSRPID	PID number	(RC)
ADSRPIDL	PID release level	(RC)
ADSRCDSC	Text description	
ADSRRET	Return Code	(RS)
ADSRREA	Reason Code	(RS)
ADSRPRID	Problem Identifier	
ADSRID	Subsystem identifier	

Notes:

1. This section has a fixed length of 100 bytes, and cannot be truncated. Optional fields (not marked with RC, RS, or RA) will appear as zero if no values are provided.

2. ADSRCID is the component ID of the application that incurred the error.

   Under some circumstances, there can be more than one component ID involved. For ADSRCID, select the component ID that is most indicative of the source of the error. The default is the component ID of the detecting program. In no case should the component ID represent a program that only administratively handles the symptom record. Additional and clarifying data (such as, other component ID involved) is optional, and may be placed in optional entries such as ADSRCDSC of section 2.1, section 4, or section 5.

   For example: if component A receives a program check; control is taken by component B, which is the program check handler. Component C provides a service to various programs by issuing SYMREC for them. In this case, the component ID of A should be given. Component B is an error handler that is unrelated to the source of the error. Component C is only an administrator. Note that, in this example, it was possible for B to know A was the program in control and the source of the program check. This precise definition is not always possible. B is the detector, and the true source of the symptom record. If the identity of A was unknown, then B would supply, as a default, its own component ID.

   ADSRCID is not a required field in this section, although it is required in section 3 after the PIDS/ prefix of the symptom string. Repeating this value in section 2.1 is desirable but not required. Where the component ID is not given in section 2.1, this field must contain zeroes.

   ADSRPID is the program identification number assigned to the program that incurred the error. ADSRPID must be provided only by IBM programs that do not have an assigned component ID. Therefore, ADSRCID contains hex zeroes if ADSRPID is provided.

3. ADSRLVL is the release level of the component indicated in ADSRCID.

4. ADSRPIDL is the release level of the program designated by ADSRPID, and it should be formatted using the characters, V, R, and M as separators, where V, R, and M represent the version, release, and modification level respectively. For example, V1R21bbbbb is Version 1 Release 2.1 without any modification. No punctuation can appear in this field, and ADSRPIDL must be provided only when ADSRPID is provided.

5. ADSRPTF is the service level. It may differ from ADSRLVL because the program may be at a higher level than the release. ADSRPTF can contain any number indicative of the service level. For example, a PTF, FMID, APAR number, or user modification number. This field is not required, but it should be provided if possible.
6. ADSRCDSC is a 32-byte area that contains text, and it is only provided at the discretion of the reporting component. It provides clarifying information.
7. ADSRREA is the reason code, and ADSRRET is the return code from the execution of SYMREC. SYMREC places these values in registers 0 and 15 and in these two fields as part of its execution. The fields are right justified, and identical to the contents of registers 0 and 15.
8. ADSRCRL is the architectural level of the record. Note that ADSRARID (section 2) is the architectural level of the SYMREC service.
9. ADSRPRID is a value that can be used to associate the symptom record with other symptom records. This value must be in EBCDIC, but it is not otherwise restricted.
10. ADSRNIBM is a flag indicating that a non-IBM program originated the symptom record.
11. ADSRSSID is the name of a subsystem. The primary purpose of this field is to allow subsystems to intercept the symptom record from programs that run on the subsystem. They may then add their own identification in this field as well as additional data in sections 4 and 5. The subsystem can then pass the symptom record to the system via SYMREC. A zero value is interpreted as no name.

Programming Notes for Section 3

Section 3 of the symptom record contains the primary symptoms associated with the error, and is provided by the application that incurred the error, or some program that acts on its behalf. The internal format of the data in section 3 is the SDB format, with a blank separating each entry. Once this data has been passed to SYMREC by the invoker, it may not be added to or modified without setting ADSRPMOD to '1'. The data in this section is EBCDIC, and no hex zeros may appear. The symptoms are in the form K/D where K is a keyword of 1 to 8 characters and D is at least 1 character. D can only be an alphanumeric or @, $, and #.

Notes:

1. The symptom K/D can have no imbedded blanks, but the '#' can be used to substitute for desired blanks. Each symptom (K/D) must be separated by at least one blank. The first symptom may start at ADSRRSCS with no blank, but the final symptom must have at least one trailing blank. The total length of each symptom (K/D combination) can not exceed 15 characters.
2. This section is provided by the component that reports the failure to the system. Once a SYMREC macro is issued, the reported information will not be added to or modified, even if the information is wrong. It is the basis for automated searches, and even poorly chosen information will compare correctly in such processing because the component consistently produces the same symptoms regardless of what information was chosen.
3. The PIDS/ entry is required, with the component ID following the slash. It is required from all programs that originate a symptom record and have component a ID assigned. Further, it must be identical to the value in ADSRCID (section 2.1) if that is provided. (ADSRCID is not a required field).

Programming Notes for Section 4

Section 4 of the symptom record contains the secondary symptoms associated with the error incident, and it is provided by the application that incurred the error, or some program that acts in its behalf. The internal format of the data in section 4 is the SDB format, with a single blank separating each entry. Once this data has been passed to SYMREC by the invoker, it may not be added to or modified without setting ADSRSMOD to 1.

Programming Notes for Section 5

Section 5 of the symptom record contains logical segments of data that are provided by the component or some program that acts in its behalf. The component stores data in section 5 before SYMREC is invoked.

Notes:

1. The first segment must be stored at symbolic location ADSR5ST. In each segment, the first two characters are a hex key value, and the second two characters are the length of the data string, which must immediately follow the two-byte length field. Adjacent segments must be packed together. The length of section 5 is in the ADSRRONL field in section 2, and this field should be correctly updated as a result of all additions or deletions to section 5.
2. There are 64K key values grouped in thirteen ranges representing thirteen potential SYMREC user categories. The data type (that is, hexadecimal, EBCDIC, etc.) of section 5 is indicated by the category of the key value. Thus, the key value indicates both the user category and the data type that are associated with the information in section 5. Because the component ID is a higher order qualifier of the key, it is only necessary to control the assignment of keys within each component ID or, if a component ID not assigned, within each PID number.

Key Value	User Category and Data Type
0001-00FF	Reserved
0100-0FFF	MVS system programs
1000-18FF	VM system programs
1900-1FFF	DOS/VSE system programs
2000-BFFF	Reserved
C000-CFFF	Product programs and non-printable hex data
D000-DFFF	Product programs and printable EBCDIC data
E000-EFFF	Reserved
F000	Any program and printable EBCDIC data
F001-F0FF	Not assignable
F100-FEFF	Reserved
FF00	Any program and non-printable hex data
FF01-FFFF	Not assignable

Chapter 9. Virtual Storage Management

You use the virtual storage area assigned to your job step by making implicit and explicit requests for virtual storage. (In addition to the virtual storage requests that you make, the system also can request virtual storage to contain the control blocks required to manage your tasks.)

Some macros represent implicit requests for storage. For example, when you invoke LINK to pass control to another load module, the system allocates storage before bringing the load module into your job pack area.

A GETMAIN or STORAGE macro is an explicit request for virtual storage. When you make an explicit storage request, the system allocates to your task the number of virtual storage bytes that you request. The macros also allow you to specify where the central storage that backs the virtual storage resides; either above or below 16 megabytes.

The CPOOL macro and callable cell pool services are also explicit requests for storage. The macro and the services provide an area called a **cell pool** from which you can obtain **cells** of storage. "Using the CPOOL Macro" on page 9-5 and Chapter 10, "Callable Cell Pool Services" describe how you can create and manage cell pools.

The DSPSERV macro is an explicit request for virtual storage that is not part of your address space. It is available for storing data, but not executing code. The two kinds of data-only spaces are **data spaces** and **hiperspaces**. For information on how to obtain and manage these virtual storage areas, see Chapter 13, "Data Spaces and Hiperspaces."

Note: If your job step is to be executed as a nonpageable (V=R) task, the REGION parameter value specified on the JOB or EXEC statement determines the amount of virtual (real) storage reserved for the job step. If you run out of storage, increase the REGION parameter size.

This chapter describes techniques you can use to obtain and release virtual storage and make efficient use of the virtual storage area reserved for your job step.

Explicit Requests for Virtual Storage

To explicitly request virtual storage, issue a GETMAIN or a STORAGE macro. When you make an explicit request, the system satisfies the request by allocating a part of the virtual storage area reserved for the job step. The virtual storage area is usually not set to zero when allocated. (The system sets storage to zero only when it initially assigns a frame to a virtual storage page.)

You explicitly release virtual storage by issuing a FREEMAIN macro or a STORAGE macro. For information about using these macros, see "Releasing Storage Through the FREEMAIN and STORAGE Macros" on page 9-5.

Specifying the Size of the Area

Virtual storage areas are always allocated to the task in multiples of eight bytes and may begin on either a doubleword or page boundary. You request virtual storage in bytes; if the number you specify is not a multiple of eight, the system rounds it up to the next higher multiple of eight. You can make repeated requests for a small number of bytes as you need the area, or you can make one large request to completely satisfy the requirements of the task. There are two reasons for making one large request. First, it is the only way you can be sure to get contiguous virtual storage and avoid fragmenting your address space. Second, you make only one request, and thus minimize the amount of system overhead.

Obtaining Storage Through the GETMAIN Macro

There are several methods of explicitly requesting virtual storage using a GETMAIN macro. Each method, which you select by coding a parameter, has certain advantages.

You can specify the actual location (above or below 16 megabytes) of the virtual area allocated by using the LOC parameter. (LOC is valid only with the RU, RC, VRU, and VRC parameters.) If you code LOC=ANY and indicate a subpool that is supported above 16 megabytes, GETMAIN tries to allocate the virtual storage area above 16 megabytes. If it cannot, or if the subpool is not supported above 16 megabytes, GETMAIN allocates the area from virtual storage below 16 megabytes.

The last three methods do not produce reenterable coding unless coded in the list and execute forms. (See "Implicit Requests for Virtual Storage" on page 9-9 for additional information.) When you use the last three types, you can allocate storage below 16 megabytes only.

The methods and the characters associated with them follow:

1. **Register Type**: There are several kinds of register requests. In each case the address of the area is returned in register 1. All of the register requests produce reenterable code because the parameters are passed to the system in registers, not in a parameter list. The register requests are as follows:

R	specifies a request for a single area of virtual storage of a specified length, located below 16 megabytes.
RU or RC	specifies a request for a single area of virtual storage of a specified length, located above or below 16 megabytes according to the LOC parameter.
VRU or VRC	specifies a request for a single area of virtual storage with length between two values that you specify, located above or below 16 megabytes according to the LOC parameter. GETMAIN attempts to allocate the maximum length you specify. If not enough storage is available to allocate the maximum length, GETMAIN allocates the largest area with a length between the two values that you specified. GETMAIN returns the length in register 0.

2. **Element Type:** EC or EU specifies a request for a single area of virtual storage, below 16 megabytes, of a specified length. GETMAIN places the address of the allocated area in a fullword that you supply.
3. **Variable Type:** VC or VU specifies a request for a single area of virtual storage below 16 megabytes with a length between two values you specify. GETMAIN attempts to allocate the maximum length you specify; if not enough storage is available to allocate the maximum length, the largest area with a length

between the two values is allocated. GETMAIN places the address of the area and the length allocated in two consecutive fullwords that you supply.

4. **List Type:** LC or LU specifies a request for one or more areas of virtual storage, below 16 megabytes, of specified lengths.

The LOC parameter also allows you to indicate whether the real frames that back the virtual storage are above or below 16 megabytes. For more information, see the description of the GETMAIN macro in *Assembler Programming Reference*.

In combination with these methods of requesting virtual storage, you can designate the request as conditional or unconditional. If the request is unconditional and sufficient virtual storage is not available to fill the request, the active task is abnormally terminated. If the request is conditional, however, and insufficient virtual storage is available, a return code of 4 is provided in register 15; a return code of 0 is provided if the request was satisfied.

Figure 9-1 shows an example of using the GETMAIN macro. The example assumes a program that operates most efficiently with a work area of 16,000 bytes, with a fair degree of efficiency with 8,000 bytes or more, inefficiently with less than 8,000 bytes. The program uses a reentrant load module having an entry name of REENTMOD, and will use it again later in the program; to save time, the load module was brought into the job pack area using a LOAD macro so that it will be available when it is required.

```
            .
            .
            GETMAIN EC,LV=16000,A=ANSWADD  Conditional request for 16,000
                                           bytes in central storage
            LTR     15,15                  Test return code
            BZ      PROCEED1               If 16,000 bytes allocated, proceed
            DELETE  EP=REENTMOD            If not, delete module and try
            GETMAIN VU,LA=SIZES,A=ANSWADD  to get less virtual storage
            L       4,ANSWADD+4            Load and test allocated length
            CH      4,MIN                  If 8,000 or more, use procedure 1
            BNL     PROCEED1               If less than 8,000 use procedure 2
PROCEED2    ...
PROCEED1    ...
MIN         DC      H'8000'                Min. size for procedure 1
SIZES       DC      F'4000'                Min. size for procedure 2
            DC      F'16000'               Size of area for maximum efficiency
ANSWADD     DC      F'0'                   Address of allocated area
            DC      F'0'                   Size of allocated area
```

Figure 9-1. Example of Using the GETMAIN Macro

The code shown in Figure 9-1 makes a conditional request for a single element of storage with a length of 16,000 bytes. The return code in register 15 is tested to determine if the storage is available; if the return code is 0 (the 16,000 bytes were allocated), control is passed to the processing routine. If sufficient storage is not available, an attempt to obtain more virtual storage is made by issuing a DELETE macro to free the area occupied by the load module REENTMOD. A second GETMAIN macro is issued, this time an unconditional request for an area between 4,000 and 16,000 bytes in length. If the minimum size is not available, the task is abnormally terminated. If at least 4,000 bytes are available, the task can continue.

The size of the area actually allocated is determined, and one of the two procedures (efficient or inefficient) is given control.

Obtaining Storage Through the STORAGE Macro

There are several ways of explicitly requesting virtual storage through the STORAGE macro with the OBTAIN parameter. In the most simple request, you issue the macro giving the length of storage you want and accepting the defaults for the optional parameters. This request is as follows:

```
STORAGE OBTAIN,LENGTH=length
```

When you issue this macro, the system uses certain defaults. The following list summarizes the defaults for optional parameters and identifies the parameters that override the system defaults.

- After STORAGE executes, you will find the address of the storage in register 1 (ADDR parameter).
- The storage is located in subpool 0 (SP parameter).
- The storage is aligned on a doubleword boundary (BNDRY parameter).
- After the macro executes, you will find the return code in register 15 (RTCD parameter).
- Whether the storage is located above or below 16 megabytes depends on the location of the caller (LOC parameter). For example, if the caller is above 16 megabytes, the virtual storage and the real frames that back the virtual storage will also be above 16 megabytes.
- The request for storage is unconditional (COND parameter). If the request is unconditional and sufficient virtual storage is not available to fill the request, the system abends the active task.

The SP, BNDRY, and COND parameters on STORAGE OBTAIN provide the same function as the SP, BNDRY and COND parameters on GETMAIN.

To make a variable length request for storage, use the LENGTH=*(maximum length, minimum length)* parameter. The maximum, which is limited by the REGION parameter on the JOB or EXEC JCL statement, is the storage amount you would prefer. The minimum is the smallest amount you can tolerate.

To specify where the virtual and central storage comes from, use the LOC parameter. You can specify that the storage be above or below 16 megabytes or in the same location as the caller. You can specify where the central storage that backs the virtual storage comes from. The LOC parameter on STORAGE is similar to the LOC parameter on GETMAIN with the RU and RC parameters that are described in "Obtaining Storage Through the GETMAIN Macro" on page 9-2. On LOC, you specify the location of the storage (both the virtual storage **and** the central storage that backs the virtual).

To request storage conditionally, use COND =YES. If the request is conditional and insufficient virtual storage is available, the system returns a code of 4 in register 15 or the location you specify on the RTCD parameter. If the system is able to satisfy the request, it returns a code of 0.

The system returns the address of the storage in the location specified by the ADDR parameter or, by default, to register 1.

The STORAGE macro is described in *Assembler Programming Reference*. The macro description includes several examples of how to use the STORAGE macro.

Releasing Storage Through the FREEMAIN and STORAGE Macros

You release virtual storage by issuing a FREEMAIN macro or a STORAGE macro with the RELEASE parameter. Neither request releases the area from control of the job step but does make the area available to satisfy the requirements of additional requests for any task in the job step. The virtual storage assigned to a task is also released when the task terminates, except as indicated under "Subpool Handling" on page 9-6. Implicit releasing of virtual storage is described under "Freeing of Virtual Storage" on page 9-12.

To release storage with the STORAGE macro, specify the amount, the address, and the subpool (SP parameter). If you are releasing all of the storage in a subpool, you can issue the SP parameter without specifying the length and the address. Releasing all of the storage in a subpool is called a **subpool release.**

Using the CPOOL Macro

The cell pool macro (CPOOL) provides programs with another way of obtaining virtual storage. This macro provides centralized, high-performance cell management services.

What is a cell pool? A cell pool is a block of virtual storage that is divided into smaller, fixed-size blocks of storage, called cells. You specify the size of the cells. You then can request cells of storage from this cell pool as you need them. If the request for cells exceeds the storage available in the cell pool, you can increase the size of the cell pool.

The CPOOL macro allows you to:

- Create a cell pool (BUILD), where all cells have the size you specify
- Obtain a cell from a cell pool if storage is available (GET,COND)
- Obtain a cell from a cell pool and extend the cell pool if storage is not available (GET,UNCOND)
- Return a cell to the cell pool (FREE)
- Free all storage for a cell pool (DELETE)
- Place the starting and ending addresses of the cell pool extents in a buffer (LIST)

You can also create and manage cell pools by using callable cell pool services. These services offer advantages over using CPOOL in some cases. Chapter 10, "Callable Cell Pool Services" describes these services. "Comparison of CPOOL Macro and Callable Cell Pool Services" on page 10-1 can help you decide whether to use the callable cell pool services or the CPOOL macro.

Subpool Handling

The system provides subpools of virtual storage to help you manage virtual storage and communicate between tasks in the same job step. Because the use of subpools requires some knowledge of how the system manages virtual storage, a discussion of virtual storage control is presented here.

Virtual Storage Control: When the job step is given a region of virtual storage in the private area of an address space, all of the storage area available for your use within that region is unassigned. Subpools are created only when a GETMAIN, STORAGE, or CPOOL macro is issued designating a subpool number (other than 0) not previously specified. If no subpool number is designated, the virtual storage is allocated from subpool 0, which is created for the job step by the system when the job-step task is initiated.

For purposes of control and virtual storage protection, the system considers all virtual storage within the region in terms of 4096-byte blocks. These blocks are assigned to a subpool, and space within the blocks is allocated to a task by the system when requests for virtual storage are made. When there is sufficient unallocated virtual storage within any block assigned to the designated subpool to fill a request, the virtual storage is allocated to the active task from that block. If there is insufficient unallocated virtual storage within any block assigned to the subpool, a new block (or blocks, depending on the size of the request) is assigned to the subpool, and the storage is allocated to the active task. The blocks assigned to a subpool are not necessarily contiguous unless they are assigned as a result of one request. Only blocks within the region reserved for the associated job step can be assigned to a subpool.

Figure 9-2 is a simplified view of a virtual storage region containing four 4096-byte blocks of storage. All the requests are for virtual storage from subpool 0. The first request from some task in the job step is for 1008 bytes; the request is satisfied from the block shown as Block A in the figure. The second request, for 4000 bytes, is too large to be satisfied from the unused portion of Block A, so the system assigns the next available block, Block B, to subpool 0, and allocates 4000 bytes from Block B to the active task. A third request is then received, this time for 2000 bytes. There is enough area in Block A (blocks are checked in the order first in, first out), so an additional 2000 bytes are allocated to the task from Block A. All blocks are searched for the closest fitting free area which will then be assigned. If the request had been for 96 bytes or less, it would have been allocated from Block B. Because all tasks may share subpool 0, Request 1 and Request 3 do not have to be made from the same task, even though the areas are contiguous and from the same 4096 byte block. Request 4, for 6000 bytes, requires that the system allocate the area from 2 contiguous blocks which were previously unassigned, Block D and Block C. These blocks are assigned to subpool 0.

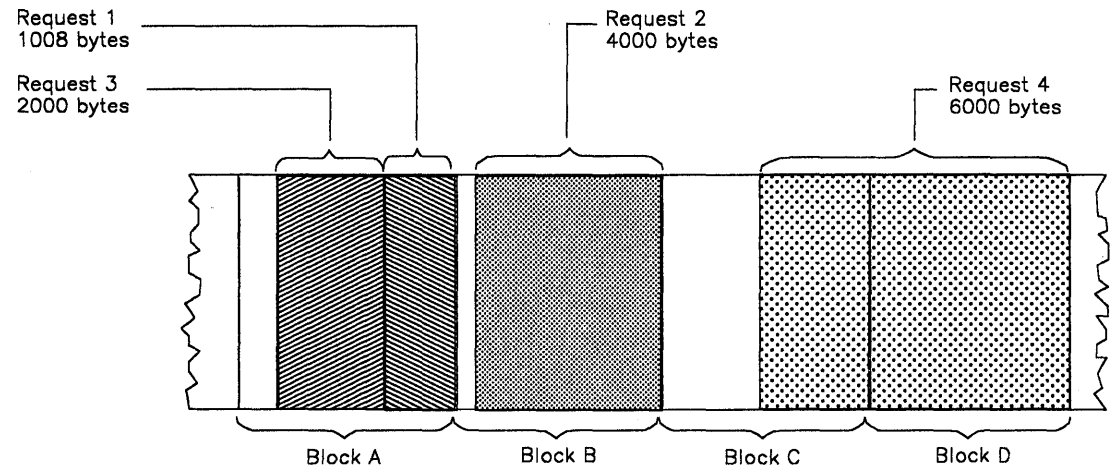


Figure 9-2. Virtual Storage Control

As indicated in the preceding example, it is possible for one 4096-byte block in subpool 0 to contain many small areas allocated to many different tasks in the job step, and it is possible that numerous blocks could be split up in this manner. Areas acquired by a task other than the job step task are not released automatically on task termination. Even if FREEMAIN or STORAGE RELEASE macros were issued for each of the small areas before a task terminated, the probable result would be that many small unused areas would exist within each block while the control program would be continually assigning new blocks to satisfy new requests. To avoid this situation, you can define subpools for exclusive use by individual tasks.

Any subpool can be used exclusively by a single task or shared by several tasks. Each time that you create a task, you can specify which subpools are to be shared. Unlike other subpools, subpool 0 is shared by a task and its subtask, unless you specify otherwise. When subpool 0 is not shared, the system creates a new subpool 0 for use by the subtask. As a result, both the task and its subtask can request storage from subpool 0 but both will not receive storage from the same 4096-byte block. When the subtask terminates, its virtual storage areas in subpool 0 are released; since no other tasks share this subpool, complete 4096-byte blocks are made available for reallocation.

Note: If the storage is shared, it is not released until the owning task terminates.

When there is a need to share subpool 0, you can define other subpools for exclusive use by individual tasks. When you first request storage from a subpool other than subpool 0, the system assigns new 4096-byte blocks to that subpool, and allocates storage from that block. The task that is then active is assigned ownership of the subpool and, therefore, of the block. When additional requests are made by the same task for the same subpool, the requests are satisfied by allocating areas from that block and as many additional blocks as are required. If another task is active when a request is made with the same subpool number, the system assigns a new block to a new subpool, allocates storage from the new block, and assigns ownership of the new subpool to the second task.

A task can specify subpools numbered from 0 to 127. FREEMAIN or STORAGE RELEASE macros can be issued to release any complete subpool except subpool 0, thus releasing complete 4096-byte blocks. When a task terminates, its unshared subpools are released automatically.

Owning and Sharing: A subpool is initially owned by the task that was active when the subpool was created. The subpool can be shared with other tasks, and ownership of the subpool can be assigned to other tasks. The macros used to handle subpools are STORAGE, GETMAIN, ATTACH and ATTACHX. In the GETMAIN macro, the SP parameter can be written to request storage from subpools 0 to 127; if you omit this parameter, the system assumes subpool 0. The parameters that deal with subpools in the ATTACH and ATTACHX macros are:

- GSPV and GSPL, which give ownership of one or more subpools (other than subpool 0) to the task being created.
- SHSPV and SHSPL, which share ownership of one or more subpools (other than subpool 0) with the new subtask.
- SZERO, which determines whether subpool 0 is shared with the subtask.

All of these parameters are optional. If they are omitted, no subpools are given to the subtask, and only subpool 0 is shared.

Creating a Subpool: If the subpool specified does not exist for the active task, a new subpool is created whenever SHSPV or SHSPL is coded on ATTACH or ATTACHX, or when a GETMAIN or STORAGE macro is issued. A new subpool zero is created for the subtask if SZERO=NO is specified on ATTACH or ATTACHX. If one of the ATTACH or ATTACHX parameters that specifies shared ownership of a subpool causes the subpool to be created, the subpool number is entered in the list of subpools owned by the task, but no blocks are assigned and no storage is actually allocated. If a GETMAIN or STORAGE macro results in the creation of a subpool, the subpool number is assigned to one or more 4096-byte blocks, and the requested storage is allocated to the active task. In either case, ownership of the subpool belongs to the active task; if the subpool is created because of an ATTACH or ATTACHX macro, ownership is transferred or retained depending on the parameter used.

Transferring Ownership: An owning task gives ownership of a subpool to a direct subtask by using the GSPV or GSPL parameters on ATTACH or ATTACHX issued when that subtask is created. Ownership of a subpool can be given to any subtask of any task, regardless of the control level of the two tasks involved and regardless of how ownership was obtained. A subpool cannot be shared with one or more subtasks and then transferred to another subtask, however; an attempt to do this results in abnormal termination of the active task. Ownership of a subpool can only be transferred if the active task has sole ownership; if the active task is sharing a subpool and an attempt is made to transfer it to a subtask, the subtask receives shared control and the originating task relinquishes the subpool. Once ownership is transferred to a subtask or relinquished, any subsequent use of that subpool number by the originating task results in the creation of a new subpool. When a task that has ownership of one or more subpools terminates, all of the virtual storage areas in those subpools are released. Therefore, the task with ownership of a subpool should not terminate until all tasks or subtasks sharing the subpool have completed their use of the subpool.

If GSPV or GSPL specifies a subpool that does not exist for the active task, no action is taken.

Sharing a Subpool: A task can share ownership of a subpool with a subtask that it attaches. Subtasks cannot share ownership of a subpool with the task that caused the attach. A program shares ownership by specifying the SHSPV or SHSPL parameters on the ATTACH or ATTACHX macro issued when the subtask is created.

Any task with ownership or shared control of the subpool can add to or reduce the size of the subpool through the use of the GETMAIN, FREEMAIN, or STORAGE macros. When a task that has shared control of the subpool terminates, the subpool is not affected.

Subpools in Task Communication: The advantage of subpools in virtual storage management is that, by assigning separate subpools to separate subtasks, the breakdown of virtual storage into small fragments is reduced. An additional benefit from the use of subpools can be realized in task communication. A subpool can be created for an originating task and all parameters to be passed to the subtask placed in the subpool. When the subtask is created, the ownership of the subpool can be passed to the subtask. After all parameters have been acquired by the subtask, a FREEMAIN or STORAGE RELEASE macro can be issued, under control of the subtask, to release the subpool virtual storage areas. In a similar manner, a second subpool can be created for the originating task, to be used as an answer area in the performance of the subtask. When the subtask is created, the subpool ownership would be shared with the subtask. Before the subtask is terminated, all parameters to be passed to the originating task are placed in the subpool area; when the subtask is terminated, the subpool is not released, and the originating task can acquire the parameters. After all parameters have been acquired for the originating task, a FREEMAIN or STORAGE RELEASE macro again makes the area available for reuse.

Implicit Requests for Virtual Storage

You make an implicit request for virtual storage every time you issue LINK, LINKX, LOAD, ATTACH, ATTACHX, XCTL or XCTLX. In addition, you make an implicit request for virtual storage when you issue an OPEN macro for a data set. This section discusses some of the techniques you can use to cut down on the amount of central storage required by a job step, and the assistance given you by the system.

Reenterable Load Modules

A reenterable load module does not modify itself. Only one copy of the load module is loaded to satisfy the requirements of any number of tasks in a job step. This means that even though there are several tasks in the job step and each task concurrently uses the load module, the only central storage needed is an area large enough to hold one copy of the load module (plus a few bytes for control blocks). The same amount of central storage would be needed if the load module were serially reusable; however, the load module could not be used by more than one task at a time.

Note: If your module is reenterable or serially reusable, the load module must be link edited, with the desired attribute, into a library. The default linkage editor attributes are nonreenterable and nonreusable.

Reenterable Macros

All of the macros described in this manual can be written in reenterable form. These macros are classified as one of two types: macro that pass parameters in registers 0 and 1, and macros that pass parameters in a list. The macros that pass parameters in registers present no problem in a reenterable program; when the macro is coded, the required parameter values should be contained in registers. For example, the POST macro requires that the ECB address be coded as follows:

```
POST ecb address
```

One method of coding this in a reenterable program would be to require this address to refer to a portion of storage that has been allocated to the active task through the use of a GETMAIN macro. The address would change for each use of the load module. Therefore, you would load one of the general registers 2-12 with the address, and designate that register when you code the macro. If register 4 contains the ECB address, the POST macro is written as follows:

```
POST (4)
```

The macros that pass parameters in a list require the use of special forms of the macro when used in a reenterable program. The macros that pass parameters in a list are identified within their descriptions in *Assembler Programming Reference.* The expansion of the standard form of these macros results in an in-line parameter list and executable instructions to branch around the list, to save parameter values in the list, to load the address of the list, and to pass control to the required system routine. The expansions of the list and execute forms of the macro simply divide the functions provided in the standard form expansion: the list form provides only the parameter list, and the execute form provides executable instructions to modify the list and pass control. You provide the instructions to load the address of the list into a register.

The list and execute forms of a macro are used in conjunction to provide the same services available from the standard form of the macro. The advantages of using list and execute forms are as follows:

- Any parameters that remain constant in every use of the macro can be coded in the list form. These parameters can then be omitted in each of the execute forms of the macro which use the list. This can save appreciable coding time when you use a macro many times. (Any exceptions to this rule are listed in the description of the execute form of the applicable macro.)
- The execute form of the macro can modify any of the parameters previously designated. (Again, there are exceptions to this rule.)
- The list used by the execute form of the macro can be located in a portion of virtual storage assigned to the task through the use of the GETMAIN macro. This ensures that the program remains reenterable.

Figure 9-3 shows the use of the list and execute forms of a DEQ macro in a reenterable program. The length of the list constructed by the list form of the macro is obtained by subtracting two symbolic addresses; virtual storage is allocated and the list is moved into the allocated area. The execute form of the DEQ macro does not modify any of the parameters in the list form. The list had to be moved to allocated storage because the system can store a return code in the list when RET=HAVE is coded. Note that the coding in a routine labeled MOVERTN is valid for lengths up to 256 bytes only. Some macros do produce lists greater than 256 bytes when many parameters are coded (for example, OPEN and CLOSE with many data control blocks, or ENQ and DEQ with many resources), so in actual practice a length check should be made. The move long instruction (MVCL) should be considered for moving large data lists.

```
            .
            .
            LA      3,MACNAME         Load address of list form
            LA      5,NSIADDR         Load address of end of list
            SR      5,3               Length to be moved in register 5
            BAL     14,MOVERTN        Go to routine to move list
            DEQ     ,MF=(E,(1))       Release allocated resource
            .
            .
* The MOVERTN allocates storage from subpool 0 and moves up to 256
* bytes into the allocated area.  Register 3 is from address,
* register 5 is length.  Area address returned in register 1.
MOVERTN     GETMAIN R,LV=(5)
            LR      4,1               Address of area in register 4
            BCTR    5,0               Subtract 1 from area length
            EX      5,MOVEINST        Move list to allocated area
            BR      14                Return
MOVEINST    MVC     0(0,4),0(3)
            .
            .
MACNAME     DEQ     (NAME1,NAME2,8,SYSTEM),RET=HAVE,MF=L
NSIADDR     ...     ...
NAME1       DC      CL8'MAJOR'
NAME2       DC      CL8'MINOR'
```

Figure 9-3. Using the List and the Execute Forms of the DEQ Macro

Nonreenterable Load Modules

The use of reenterable load modules does not automatically conserve virtual storage; in many applications it will actually prove wasteful. If a load module is not used in many jobs and if it is not employed by more than one task in a job step, there is no reason to make the load module reenterable. The allocation of virtual storage for the purpose of moving coding from the load module to the allocated area is a waste of both time and virtual storage when only one task requires the use of the load module.

You do not need to make a load module reenterable or serially reusable if reusability is not really important to the logic of your program. Of course, if reusability is important, you can issue a LOAD macro to load a reusable module, and later issue a DELETE macro to release its area.

Notes:

1. If your module is reenterable or serially reusable, the load module must be link edited, with the desired attribute, into a library. The default linkage editor attributes are nonreenterable and nonreusable.
2. A module that does not modify itself (a refreshable module) reduces paging activity because it does not need to be paged out.

Freeing of Virtual Storage

The system establishes two responsibility counts for every load module brought into virtual storage in response to your requests for that load module. The responsibility counts are lowered as follows:

- If the load module was requested in a LOAD macro, that responsibility count is lowered when using a DELETE macro.
- If the load module was requested on LINK, LINKX, ATTACH, ATTACHX, XCTL, or XCTLX, that responsibility count is lowered when using XCTL or XCTLX or by returning control to the system.
- When a task is terminated, the responsibility counts are lowered by the number of requests for the load module made by LINK, LINKX, LOAD, ATTACH, ATTACHX, XCTL, or XCTLX during the performance of that task, minus the number of deletions indicated above.

The virtual storage area occupied by a load module is released when the responsibility counts reach zero. When you plan your program, you can design the load modules to give you the best trade-off between execution time and efficient paging. If you use a load module many times in the course of a job step, issue a LOAD macro to bring it into virtual storage; do not issue a DELETE macro until the load module is no longer needed. Conversely, if a load module is used only once during the job step, or if its uses are widely separated, issue LINK or LINKX to obtain the module and issue an XCTL or XCTLX from the module (or return control to the system) after it has been executed.

There is a minor problem involved in the deletion of load modules containing data control blocks (DCBs). An OPEN macro instruction must be issued before the DCB is used, and a CLOSE macro issued when it is no longer needed. If you do not issue a CLOSE macro for the DCB, the system issues one for you when the task is terminated. However, if the load module containing the DCB has been removed from virtual storage, the attempt to issue the CLOSE macro causes abnormal termination of the task. You must either issue the CLOSE macro yourself before deleting the load module, or ensure that the data control block is still in virtual storage when the task is terminated (possibly by issuing a GETMAIN and creating the DCB in the area that had been allocated by the GETMAIN).

Chapter 10. Callable Cell Pool Services

Callable cell pool services manage areas of virtual storage in the primary address space, in data spaces and in address spaces other than the primary address space. A cell pool is an area of virtual storage that is subdivided into fixed-sized areas of storage called **cells**, where the cells are the size you specify. A cell pool contains:

- An anchor
- At least one extent
- Any number of cells, all having the same size.

The **anchor** is the starting point or foundation on which you build a cell pool. Each cell pool has only one anchor. An **extent** contains information that helps callable cell pool services manage cells and provides information you might request about the cell pool. A cell pool can have up to 65,536 extents, each responsible for its own cell storage. Your program determines the size of the cells and the cell storage. Figure 10-1 on page 10-3 illustrates the three parts of a cell pool.

Through callable cell pool services, you build the cell pool. You can then obtain cells from the pool. When there are no more cells available in a pool, you can use callable cell pool services to enlarge the pool.

To use callable cell pool services, your program issues the CALL macro to invoke one of the following services:

- Build a cell pool and initialize an anchor (CSRPBLD service)
- Expand a cell pool by adding an extent (CSRPEXP service)
- Connect cell storage to an extent (CSRPCON service)
- Activate previously connected storage (CSRPACT service)
- Deactivate an extent (CSRPDAC service)
- Disconnect the cell storage for an extent (CSRPDIS service)
- Allocate a cell from a cell pool (CSRPGET and CSRPRGT services)
- Return a cell to the cell pool (CSRPFRE and CSRPRFR services)
- Query the cell pool (CSRPQPL service)
- Query a cell pool extent (CSRPQEX service)
- Query a cell (CSRPQCL service).

Comparison of CPOOL Macro and Callable Cell Pool Services

Callable cell pool services are similar to the CPOOL macro, but with some additional capabilities. A program executing in any state or mode (disabled, locked, AR mode, SRB mode, etc.) can use the services to manage storage in data spaces as well as address spaces. The services allow you to define cell boundaries and to expand and contract cell pools. Another difference is in how CPOOL and the callable cell pool services handle the requests to free cells. CPOOL corrupts storage if you try to free a cell that has not been obtained (through CPOOL GET), or if you try to free a cell for a second time. Callable cell pool services accept the request, but do no processing except to return a code to your program.

The following table describes other differences; it can help you decide between the two ways to manage cell pools.

If your program:	Use:
Is in AR mode	Cell pool services. (CPOOL has mode restrictions.)
Needs to reduce the size of a cell pool	Cell pool services. (CPOOL supports expansion but not contraction.)
Needs data space storage	Cell pool services. (CPOOL supports only the primary address space.)
Needs storage in an address space other than the primary	Cell pool services. (CPOOL supports only primary address space storage.)
Must define cell boundaries	Cell pool services. (CPOOL supports only 8-byte boundaries.)
Requires high performance on GETs and FREEs	CPOOL.

In some ways, callable cell pool services require more work from the caller than CPOOL does. For example, the services require the following actions that the CPOOL macro does not require:

- Use the GETMAIN, STORAGE OBTAIN, or DSPSERV macro to obtain the storage area for the cell pool.
- Provide the beginning addresses of the anchor, the extents, and cell storage areas.
- Provide the size of each extent and the cell storage that the extent is responsible for.

Storage Considerations

The virtual storage for the cell pool must reside in an address space or a data space, and not in a hiperspace.

- The anchor and extents must reside within the same address space or data space.
- The cells must reside within one address space or data space; that space can be different from the one that contains the anchor and extents.

The following diagram illustrates the anchor and extents in Data/Address Space A and the cell storage in Data/Address Space B.

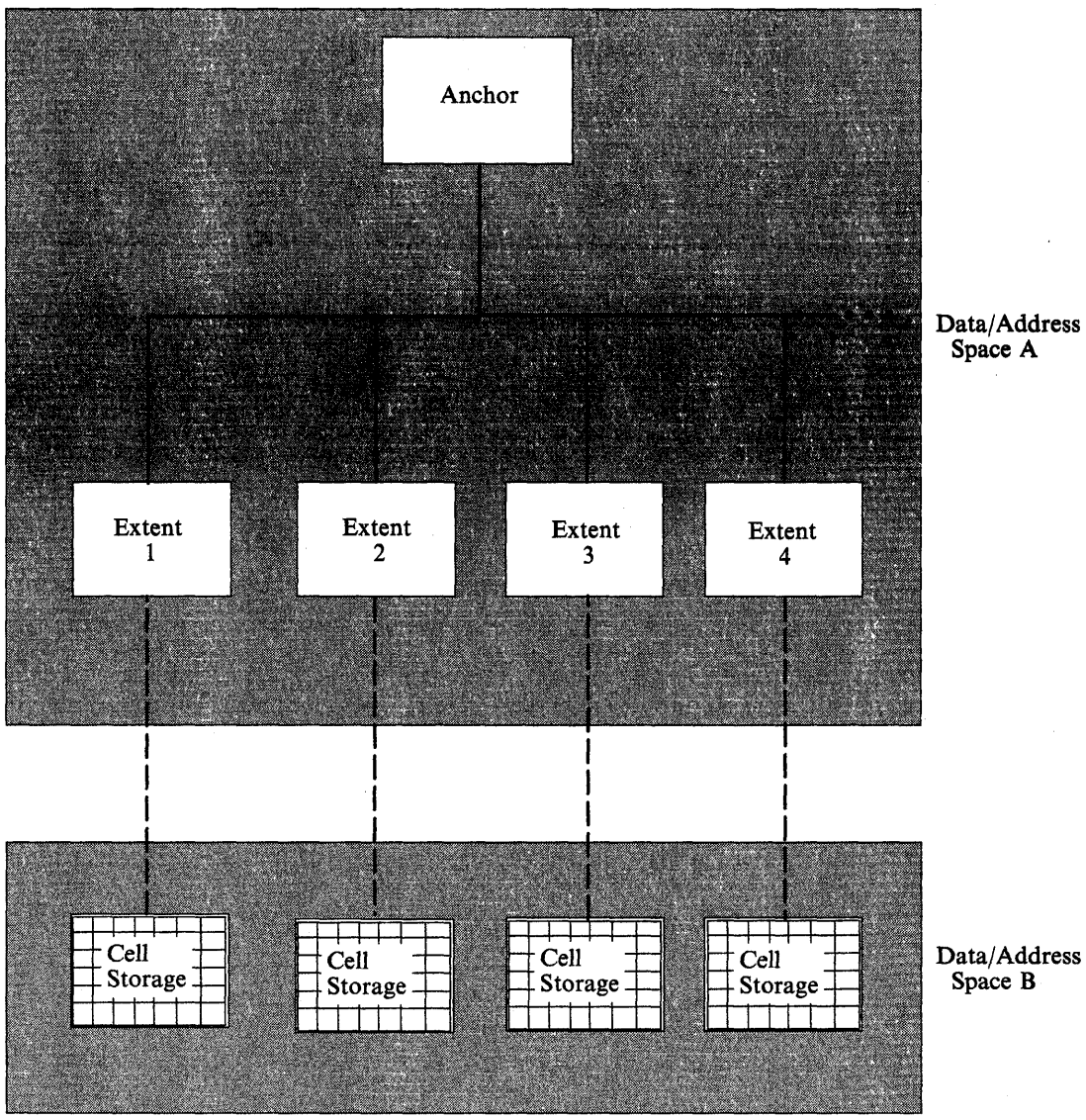


Figure 10-1. Cell Pool Storage

Before you can obtain the first cell from a cell pool, you must plan the location of the anchor, the extents, and the cell storage. You must obtain the storage for the following areas and pass the following addresses to the services:

- The anchor, which requires 64 bytes of storage
- The extent, which requires 128 bytes plus one byte for every eight cells of cell storage
- The cell storage.

When you plan the size of the cell storage, consider the total requirements of your application for this storage and some performance factors. Although a single extent may contain any number of cells (up to 2^{24} bytes, or 16,777,216), you might wish to have multiple extents for performance purposes. Avoid having a large number of extents, where each extent is responsible for a small number of cells. In general, a greater requirement for cells should mean a proportionately smaller number of extents. The following two examples illustrate this point.

If you have 10,000 cells in the pool, a good extent size is 2,500 cells per extent.

If you have 100,000 cells in the pool, a good extent size is 10,000 cells per extent.

"Using Callable Cell Pool Services to Manage Data Space Areas" on page 13-18 contains an example of using callable cell pool services with data spaces. It also describes some storage considerations.

Link-editing Callable Cell Pool Services

Any program that invokes callable cell pool services must be link-edited with an IBM-provided linkage-assist routine. The linkage-assist routine provides the logic needed to locate and invoke the callable services. The linkage-assist routine resides in SYS1.CSSLIB. The following example shows the JCL needed to link-edit a program with the linkage-assist routine.

```
//LINKJOB    JOB  'accountinfo','name',CLASS=x,
//                MSGCLASS=x,NOTIFY=userid,MSGLEVEL=(1,1),REGION=4096K
//LINKSTP1   EXEC  PGM=HEWLH096,PARM='LIST,LET,XREF,REFR,RENT,NCAL,
//                SIZE=(1800K,128K)'
//SYSPRINT   DD    SYSOUT=x
//SYSLMOD    DD    DSNAME=userid.LOADLIB,DISP=SHR
//SYSUT1     DD    UNIT=SYSDA,SPACE=(TRK,(5,2))
//SYSLIN     DD    *
  INCLUDE    OBJDD1(userpgm)
  INCLUDE    OBJDD2(CSRCPOOL)
  NAME       userpgm(R)
//OBJDD1     DD    DSN=userid.OBJLIB,DISP=SHR
//OBJDD2     DD    DSN=SYS1.CSSLIB,DISP=SHR
```

The example JCL assumes that the program you are link-editing is reentrant.

Using Callable Cell Pool Services

The following topics describe how you can use callable cell pool services to control storage and request information about the cell pools. The discussion of creating a cell pool and adding an extent assumes that you have already obtained the storage for these areas.

To create a cell pool, call the CSRPBLD service. This service initializes the anchor for the cell pool, assigns the name of the pool, and establishes the size of the cells.

To add an extent and connect it to the cell storage, call the CSRPEXP service. You need at least one extent in a cell pool. Each extent is responsible for one cell storage area. You can add an extent to increase the numbers of cells; the maximum number of extents in a cell pool is 65,536. The CSRPEXP service initializes an extent for the cell pool, connects the cell storage area to the extent, and activates the cell storage for the extent.

Having activated the cell storage for an extent, you can now process GET requests from the cells that the extent represents.

To contract a cell pool, deactivate its extents, and disconnect its storage, use the CSRPDAC and CSRPDIS services. CSRPDAC deactivates an extent and prevents the processing of any further GET requests from the storage that the extent represents. Cell FREE requests are unaffected. (You can use the CSRPACT service to reactivate an inactive extent; reactivating undoes the effect of using CSRPDAC.)

CSRPDIS disconnects the cell storage from an extent and makes cell storage unavailable. After you disconnect an extent, you can free the cell storage associated with the extent. Do not free the extent itself until you have finished using the entire pool.

To reuse a deactivated, disconnected extent, call the CSRPCON and CSRPACT services, not CSRPEXP. This is generally the only time you will need to use these two services. CSRPCON reconnects an extent to cell pool storage that you have not explicitly freed or that connects the extent to cells in newly-obtained storage. If you reconnect the extent to new cell storage, be sure that the extent is large enough to support the size of the cell storage. (CSRPCON undoes the effects of using CSRPDIS.) CSRPACT activates the cell storage for the extent. You can now issue GET requests for the cells.

To allocate (that is, obtain) cells and deallocate (that is, free) previously allocated cells, you have a choice of two forms of the same services. One service form supports the standard CALL interface. The other supports a register interface and is appropriate for programs that cannot obtain storage for a parameter list. The two service functions are identical; however, the calling interface is different.

The CSRPGET (standard CALL interface) and CSRPRGT (register interface) services allocate a cell from the cell pool. You can allocate cells only from extents that have not been deactivated. Such an extent is called an **active extent**. The services return to the caller the address of the allocated cell.

The CSRPFRE (standard CALL interface) and CSRPRFR (register interface) services return a previously allocated cell to a cell pool. They return a code to the caller if they cannot find the cell associated with an extent. If you free the last allocated cell in an inactive extent, you will receive a unique code. You may use this information to initiate cell pool contraction.

To obtain status about a cell pool, use one of three services. These services do not prevent the cell pool from changing during a status query. They return status as it is at the time you issue the CALL.

The CSRPQPL service returns information about the entire cell pool. It returns the following:

- Pool name
- Cell size
- Total number of cells in active extents
- Total number of available cells associated with active extents
- Number of extents in the cell pool.

The CSRPQEX service returns information about a specific extent. It returns the following:

- Address and length of the extent
- Address and length of the cell storage area
- Total number of cells associated with the extent
- Number of available cells associated with the extent.

The CSRPQCL service returns information about a cell. It returns the following:

- Number of the extent that represents the cell
- Cell allocation status.

Handling Return Codes

Each time you call a service, you receive a return code. The return code indicates whether the service completed successfully, encountered an unusual condition, or was unable to complete successfully.

Standard CALL interface services pass return codes in both the parameter list and register 15.

When you receive a return code that indicates a problem or an unusual condition, your program can either attempt to correct the problem, or can terminate its processing.

Callable Cell Pool Services Coding Example

The code in this example invokes callable cell pool services. The anchor, the one extent, and the cell storage are all in a data space. The caller obtains a cell from the cell storage area and requests information about the pool, the extent, and the cell. Use the example to supplement and reinforce information that is presented elsewhere in this chapter.

```
         CSRCPASM              INVOKE CELL POOL SERVICES ASSEMBLER DECLARES
         SAC   512             SET AR ASC MODE
         SYSSTATE ASCENV=AR
*
* Establish addressability to code.                                  *
*
         LAE   AR12,0
         BASR  R12,0
         USING *,R12
*
* Get data space for the cell pool.                                  *
*
GETDSP   DSPSERV CREATE,NAME=DSPCNAME,STOKEN=DSPCSTKN,                X
               BLOCKS=DSPBLCKS,ORIGIN=DSPCORG
*
* Add the data space to caller's access list.                        *
*
GETALET  ALESERV ADD,STOKEN=DSPCSTKN,ALET=DSPCALET,AL=WORKUNIT
         L     2,DSPCORG        ORIGIN OF SPACE IN GR2
         ST    2,DSPCMARK       DSPCMARK IS MARK FOR DATA SPACE
*
* Copy ALET to ANCHALET for calls to cell pool services.             *
*
         MVC   ANCHALET(4),DSPCALET
*
* Set address and size of the anchor
*
         L     R4,DSPCMARK
         ST    R4,ANCHADDR
         A     R4,ANCHSIZE
         ST    R4,DSPCMARK
*
* Call the build service.                                            *
*
         CALL CSRPBLD,(ANCHALET,ANCHADDR,USERNAME,CELLSIZE,RTNCODE)
*
* Set address and size of the extent and connect extent to cells *
*
         L     R4,DSPCMARK              RESERVES
         ST    R4,XTNTADDR
         A     R4,XTNTSIZE              SETS SIZE OF EXTENT
         ST    R4,CELLSTAD
         A     R4,CELLSTLN              SETS SIZE OF CELL STORAGE
         ST    R4,DSPCMARK              DATA
         CALL CSRPEXP,(ANCHALET,ANCHADDR,XTNTADDR,XTNTSIZE,           X
               CELLSTAD,CELLSTLN,EXTENT,RTNCODE)
*
* Get a cell.  CELLADDR receives the address of the cell.            *
*
         CALL CSRPGET,(ANCHALET,ANCHADDR,CELLADDR,RTNCODE)
*
```

```
* The program uses the cells.
*
* Query the pool, the extent, and the cell.                    *
*
        CALL CSRPQPL,(ANCHALET,ANCHADDR,QNAME,QCELLSZ,QTOT_CELLS,     X
              QAVAIL_CELLS,QNUMEXT,QRTNCODE)
        CALL CSRPQEX,(ANCHALET,ANCHADDR,EXTENT,QEXSTAT,QXTNT_ADDR,    X
              QXTNT_LEN,QCELL_ADDR,QCELL_LEN,QTOT_CELLS,               X
              QAVAIL_CELLS,QRTNCODE)
        CALL CSRPQCL,(ANCHALET,ANCHADDR,CELLADDR,QCLAVL,QCLEXT,      X
              QRTNCODE)
*
* Free the cell.                                                *
*
        CALL CSRPFRE,(ANCHALET,ANCHADDR,CELLADDR,RTNCODE)
*
* Deactivate the extent.                                        *
*
        CALL CSRPDAC,(ANCHALET,ANCHADDR,EXTENT,RTNCODE)
*
* Disconnect the extent.                                        *
*
        CALL CSRPDIS,(ANCHALET,ANCHADDR,EXTENT,QCELL_ADDR,QCELL_LEN,  X
              QRTNCODE)
*
* Remove the data space from the access list.                   *
*
        ALESERV DELETE,ALET=DSPCALET
*
* Delete the data space.                                        *
*
        DSPSERV DELETE,STOKEN=DSPCSTKN
*
* Return to caller.
*
        BR 14

*****************************************************************
* Constants and data areas used by cell pool services           *
*****************************************************************
*
CELLS_PER_EXTENT  EQU  512
EXTENTS_PER_POOL  EQU  10
CELLSIZE_EQU      EQU  256
CELLS_PER_POOL    EQU  CELLS_PER_EXTENT*EXTENTS_PER_POOL
XTNTSIZE_EQU      EQU  128+(((CELLS_PER_EXTENT+63)/64)*8)
STORSIZE_EQU      EQU  CELLS_PER_EXTENT*CELLSIZE_EQU
CELLS_IN_POOL     DC   A(CELLS_PER_POOL)
```

```
ANCHALET  DS   F
ANCHADDR  DS   F
CELLSIZE  DC   A(CELLSIZE_EQU)
USERNAME  DC   CL8'MYCELLPL'
ANCHSIZE  DC   F'64'
XTNTSIZE  DC   A(XTNTSIZE_EQU)
XTNTADDR  DS   F
CELLSTAD  DS   F
CELLSTLN  DC   A(STORSIZE_EQU)
CELLADDR  DS   F
EXTENT    DS   F
STATUS    DS   F
RTNCODE   DS   F
*
*****************************************************************
* Constant data and areas for data space                        *
*****************************************************************
*
         DS    0D
DSPCSTKN DS    CL8                     DATA SPACE STOKEN
DSPCORG  DS    F                       DATA SPACE ORIGIN RETURNED
DSPCSIZE EQU   STORSIZE_EQA*EXTENTS_PER_POOL 1.28MEG DATA SPACE
DSPBLCKS DC    A((DSPCSIZE+4095)/4096) BLOCKS FOR 10K DATA SPACE
DSPCALET DS    F
DSPCMARK DS    F                       HIGH WATER MARK FOR DATA SPACE
DSPCNAME DC    CL8'DATASPC1'           DATA SPACE NAME
*
*****************************************************************
* Values returned by queries                                    *
*****************************************************************
*
QNAME          DS   CL8
QCELLSZ        DS   F
QNUMEXT        DS   F
QEXTNUM        DS   F
QEXSTAT        DS   F
QXTNT_ADDR     DS   F
QXTNT_LEN      DS   F
QCELL_ADDR     DS   F
QCELL_LEN      DS   F
QTOT_CELLS     DS   F
QAVAIL_CELLS   DS   F
QRTNCODE       DS   F
RC             DS   F
QCLADDR        DS   F
QCLEXT         DS   F
QCLAVL         DS   F
```

Chapter 11. Data-in-Virtual

Data-in-virtual simplifies the writing of applications that use large amounts of data from permanent storage. Applications can create, read, and update data without the I/O buffer, blocksize, and record considerations that the traditional GET and PUT types of access methods require.

By using the services of data-in-virtual, certain applications that access large amounts of data can potentially improve their performance and their use of system resources. Such applications have an accessing pattern that is non-sequential and unpredictable. This kind of pattern is a function of conditions and values that are revealed only in the course of the processing. In these applications, the sequential record subdivisions of conventional access methods are meaningless to the central processing algorithm. It is difficult to adapt this class of applications to conventional record management programming techniques, which require all permanent storage access to be fundamentally record-oriented. Through the DIV macro, data-in-virtual provides a way for these applications to manipulate the data without the constraints of record-oriented processing.

An application written for data-in-virtual views its permanent storage data as a seamless body of data without internal record boundaries. By using the data-in-virtual MAP service, the application can make any portion of the object appear in virtual storage in an area called a **virtual storage window**. The window can exist in an address space, a data space, or a shared or non-shared standard hiperspace. (See "Example of Mapping a Data-in-Virtual Object to a Data Space" on page 13-22 and "Using Data-in-Virtual with Hiperspaces" on page 13-37 for more information.) When the window is in a data space, the application can reference and update the data in the window by using assembler instructions. When the window is in a hiperspace, the application uses the HSPSERV macro to reference and update the data. To copy the updates to the object, the application uses the data-in-virtual SAVE service.

The data-in-virtual services process the application data in 4096-byte (4K-byte) units on 4K-byte boundaries called blocks. The application data resides in what is called a **data-in-virtual object,** a **data object,** or simply an **object**. The data-in-virtual object is a continuous string of uninterrupted data. The data object can be either a VSAM linear data set or a non-shared standard hiperspace. Choosing a linear data set as an object or a non-shared standard hiperspace as an object depends on your application. If your application requires the object to retain data, choose a linear data set, which provides permanent storage on DASD. A hiperspace object provides temporary storage.

When to Use Data-in-Virtual

When an application reads more input and writes more output data than necessary, the unnecessary reads and writes impact performance. You can expect improved performance from data-in-virtual because it reduces the amount of unnecessary I/O.

As an example of unnecessary I/O, consider the I/O performed by an interactive application that requires immediate access to several large data sets. The program knows that some of the data, although not all of it, will be accessed. However, the program does not know ahead of time which data will be accessed. A possible strategy for gaining immediate access to all the data is to read all the data ahead of time, reading each data set in its entirety insofar as this is possible. Once read into processor storage, the data can be accessed quickly. However, if only a small percentage of the data is likely to be accessed during any given period, the I/O performed on the unaccessed data is unnecessary.

Furthermore, if the application changes some data in main storage, it might not keep track of the changes. Therefore, to guarantee that all the changes are captured, the application must then write entire data sets back onto permanent storage even though only relatively few bytes are changed in the data sets.

Whenever such an application starts up, terminates, or accesses different data sets in an alternating manner, time is spent reading data that is not likely to be accessed. This time is essentially wasted, and the amount of it is proportional to the amount of unchanged data for which I/O is performed. Such applications are suitable candidates for a data-in-virtual implementation.

Factors Affecting Performance

When you write applications using the techniques of data-in-virtual, the I/O takes place only for the data referenced and saved. If you run an application using conventional access methods, and then run it a second time using data-in-virtual techniques, you will notice a difference in performance. This difference depends on two factors: the **size** of the data set and its **access pattern** (or reference pattern). Size refers to the magnitude of the data sets that the application must process. The access pattern refers to how the application references the data.

In order to improve performance by using the data-in-virtual application, your data sets must be large **and** the pattern must be scattered throughout the data set.

Engineering and scientific applications often use data access patterns that are suitable for data-in-virtual. Among the applications that can be considered for a data-in-virtual implementation are:

- Applications that process large arrays
- VSAM relative record applications
- BDAM fixed length record applications

Commercial applications sometimes use data access patterns that are not suitable because they are predictable and sequential. If the access pattern of a proposed application is fundamentally sequential or if the data set is small, a conventional VSAM (or other sequential access method) implementation may perform better than a data-in-virtual implementation. However, this does not rule out commercial applications as data-in-virtual candidates. If the performance factors are favorable, any type of application, commercial or scientific, is suitable for a data-in-virtual implementation.

Before you can use the DIV macro to process a linear data set object or a hiperspace object, you must create either the data set or the hiperspace. Chapter 13, "Data Spaces and Hiperspaces" on page 13-1 explains how to create a hiperspace. The following section explains how to create a linear data set.

Creating a Linear Data Set

To create the data set, you need to specify the DEFINE CLUSTER function of IDCAMS with the LINEAR parameter. When you code the SHAREOPTIONS parameter for DEFINE CLUSTER, the cross-system value must be 3; that is, you may code SHAREOPTIONS as (1,3), (2,3), (3,3), or (4,3). Normally, you should use SHAREOPTIONS (1,3). However, you can use the LOCVIEW parameter of the DIV macro in conjunction with the other SHAREOPTIONS. LOCVIEW is described on page 11-8. For a complete explanation of SHAREOPTIONS, see the *VSAM Administration Guide* or *Managing VSAM Data Sets*.

The following is a sample job that invokes Access Method Services (IDCAMS) to create the linear data set named DIV.SAMPLE on the volume called DIVPAK. When IDCAMS creates the data set, it creates it as an empty data set. Note that there is no RECORDS parameter; linear data sets do not have records.

```
//JNAME    JOB 'ALLOCATE LINEAR',MSGLEVEL=(1,1),
//         CLASS=R,MSGCLASS=D,USER=JOHNDOE
//*
//*        ALLOCATE A VSAM LINEAR DATASET
//*
//CLUSTPG  EXEC PGM=IDCAMS,REGION=4096K
//SYSPRINT DD SYSOUT=*
//DIVPAK   DD UNIT=3380,VOL=SER=DIVPAK,DISP=OLD
//SYSIN    DD *
           DEFINE CLUSTER (NAME(DIV.SAMPLE) -
                  VOLUMES(DIVPAK) -
                  TRACKS(1,1) -
                  SHAREOPTIONS(1,3) -
                  LINEAR)
/*
```

For further information on creating linear VSAM data sets and altering entry-sequenced VSAM data sets, see *Integrated Catalog Administration: Access Method Services Reference* or *Summary of Access Method Services for the Integrated Catalog Facility*.

Using the Services Of Data-in-Virtual

Each invocation of the DIV macro requests any one of eight distinct services provided by data-in-virtual:

- IDENTIFY
- ACCESS
- MAP
- SAVE
- RESET
- UNMAP
- UNACCESS
- UNIDENTIFY

Identify

An application must use IDENTIFY to tell the system which data-in-virtual object it wants to process. IDENTIFY generates a unique ID, or token, that uniquely represents an application's request to use the given data object. The system returns this ID to the application. When the application requests other kinds of services with the DIV macro, the application supplies this ID to the system as an input parameter. Specify DDNAME for a linear data set object and STOKEN for a hiperspace object.

Access

To gain the right to view or update the object, an application must use the ACCESS service. You normally invoke ACCESS after you invoke IDENTIFY and before you invoke MAP. ACCESS is similar to the OPEN macro of VSAM. It has a mode parameter of READ or UPDATE, and it gives your application the right to read or update the object.

If the object is a data set and if the SHAREOPTIONS parameter used to allocate the linear data set implies serialization, the system automatically serializes your access to the object. If access is not automatically serialized, you can serialize access to the object by using the ENQ, DEQ, and the RESERVE macros. If you do not serialize access to the object, you should consider using the LOCVIEW parameter to protect your window data against the unexpected changes that can occur when access to the object is not serialized. LOCVIEW is described on page 11-8.

If the object is a hiperspace, DIV ensures that only one program can write to the object and that multiple users can only read the object. Only the task that owns the corresponding ID can issue ACCESS.

Map

The data object is stored in units of 4096-byte blocks. An application uses the MAP service to specify the part of the object that is to be processed in virtual storage. It can specify the entire object (all of the blocks), or a part of the object (any continuous range of blocks). Because parts of the same object can be viewed simultaneously through several different windows, the application can set up separate windows on the same object. However, a specific page of virtual storage cannot be in more than one window at a time.

After ACCESS, the application obtains a virtual storage area large enough to contain the window. The size of the object, which ACCESS optionally returns, can determine how much virtual storage you need to request. After requesting virtual storage, the application invokes MAP. MAP establishes a one to one correspondence between blocks in the object and pages in virtual storage. A continuous range of pages corresponds to a continuous range of blocks. This correspondence is called a **virtual storage window**, or a **window**.

After MAP, the application can look into the virtual storage area that the window contains. When it looks into this virtual storage area, it sees the same data that is in the object. When the application references this virtual storage area, it is referencing the data from the object. To reference the area in the window, the application simply uses any conventional processor instructions that access storage.

Although the object data becomes available in the window when the application invokes MAP, no actual movement of data from the object into the window occurs at that time. Actual movement of data from the object to the window occurs only when the application refers to data in the window. When the application references a page in the window for the first time, a page fault occurs. When the page fault occurs, the system reads the permanent storage block into central storage.

When the system brings data into central storage, the data movement involves only the precise block that the application references. The system updates the contents of the corresponding page in the window with the contents of the linear data set object. Thus, the system brings in only the blocks that an application references into central storage. The sole exception to the system bringing in only the referenced blocks occurs when the application specifies LOCVIEW=MAP with the ACCESS service for a data set object.

Notes:

1. If the application specifies LOCVIEW=MAP with ACCESS, the entire window is immediately filled with object data when the application invokes MAP.
2. If, when an application invokes MAP, it would rather keep in the window the data that existed before the window was established (instead of having the object data appear in the window), it can specify RETAIN=YES. Specifying RETAIN=YES is useful when creating an object or overlaying the contents of an object.
3. An important concept for data-in-virtual is the concept of **freshly obtained** storage. When virtual storage has been obtained and not subsequently modified, the storage is considered to be **freshly-obtained**. The storage is also in this state when it has been obtained as a data space by using a DSPSERV CREATE and not subsequently modified. After a DSPSERV RELEASE, the storage is still considered freshly obtained until it has been modified. When referring to this storage or any of its included pages, this book uses "freshly

obtained storage"and "freshly obtained pages". If a program stores into a freshly obtained page, only that page loses its freshly obtained-status, while other pages still retain it.

Save and Reset

After the MAP service, the application can access the data inside the window directly through normal programming techniques. When the application changes some data in the window, however, the data on the object does not consequently change. If the application wants the data changes in the window to appear in the object, it must use the SAVE service. SAVE writes all changed blocks within the range to range to be saved inside the window to the object. It does not write unchanged blocks. When SAVE completes, the object contains any changes that the application made inside the virtual storage window. If a SAVE is preceded by another SAVE, the second SAVE will pick up only the changes that occurred since the previous SAVE.

If the application changes some data in a virtual storage window but then decides not to keep those changes, it can use the RESET service to reload the window with data from the object. RESET reloads only the blocks that have been changed unless you specify or have specified RELEASE=YES.

Unmap

When the application is finished processing the part of the object that is in the window, it eliminates the window by using UNMAP. To process a different part of the object, one not already mapped, the application can use the MAP service again. The SAVE, RESET, MAP, and UNMAP services can be invoked repeatedly as required by the processing requirements of the application.

If you issued multiple MAPs to different STOKENs, use STOKEN with UNMAP to identify the data space or hiperspace you want to unmap.

Unaccess

If the application has temporarily finished processing the object but still has other processing to perform, it uses UNACCESS to relinquish access to the object. Then other programs can access the object. If the application needs to access the same object again, it can regain access to the object by using the ACCESS service again without having to use the IDENTIFY service again.

Unidentify

UNIDENTIFY ends the use of a data-in-virtual object under a previously assigned ID that the IDENTIFY service returned.

The IDENTIFY Service

Your program uses IDENTIFY to select the data-in-virtual object that you want to process. IDENTIFY has four parameters: ID, TYPE, DDNAME, and STOKEN.

The following examples show two ways to code the IDENTIFY service.

Hiperspace object:

```
DIV IDENTIFY,ID=DIVOBJID,TYPE=HS,STOKEN=HSSTOK
```

Data set object:

```
DIV IDENTIFY,ID=DIVOBJID,TYPE=DA,DDNAME=DDAREA
```

ID: The ID parameter specifies the address where the IDENTIFY service returns a unique eight-byte name that connects a particular user with a particular object. This name is an output value from IDENTIFY, and it is also a required input value to all other services.

Simultaneous requests for different processing activities against the same data-in-virtual object can originate from different tasks or from different routines within the same task or the same routine. Each task or routine requesting processing activity against the object must first invoke the identify service. To correlate the various DIV macro invocations and processing activities, the eight-byte IDs generated by IDENTIFY are sufficiently unique to reflect the individuality of the IDENTIFY request, yet they all reflect the same data-in-virtual object.

TYPE: The TYPE parameter indicates the type of data-in-virtual object, either a linear data set (TYPE=DA) or a hiperspace (TYPE=HS).

DDNAME: When you specify TYPE=DA for a data set object, you must specify DDNAME to identify your data-in-virtual object. If you specify TYPE=HS with IDENTIFY, do not specify DDNAME. (Specify STOKEN instead.)

STOKEN: When you specify TYPE=HS for a hiperspace object, you must specify STOKEN to identify that hiperspace. The STOKEN must be addressable in your primary address space. The hiperspace must be a non-shared standard hiperspace and must be owned by the task issuing the IDENTIFY. The system does not verify the STOKEN until your application uses the associated ID to access the object.

The ACCESS Service

Your program uses the ACCESS service to request permission to read or update the object. ACCESS has four parameters: ID, MODE, SIZE, and LOCVIEW.

The following example shows one way to code the ACCESS service.

```
DIV ACCESS,ID=DIVOBJID,MODE=UPDATE,SIZE=OBJSIZE
```

ID: When you issue a DIV macro that requests the ACCESS service, you must also specify, on the ID parameter, the identifier that the IDENTIFY service returned. The ID parameter tells the system what object you want access to. When you request permission to access the object under a specified ID, the system checks the following conditions before it grants the access:

- You previously established the ID specified with your ACCESS request by invoking IDENTIFY.
- You have not already accessed the object under the same unique eight-byte ID. Before you can reaccess an already-accessed object under the same ID, you must first invoke UNACCESS for that ID.
- If your installation uses RACF and the object is a linear data set, you must have the proper RACF authorization to access the object.
- If you are requesting read access, the object must not be empty. Use the MODE parameter to request read or update access.

- If the data object is a hiperspace, the system rejects the request if the hiperspace:
  - Has ever been the target of an ALESERV ADD
  - Has one or more readers **and** one updater. (That is, the hiperspace can have readers and it can have one updater, but it can't have both.)

MODE: The MODE parameter specifies how your program will access the object. You can specify a mode parameter of READ or UPDATE. They are described as follows:

- READ lets you read the object, but prevents you from using SAVE, to change the object.
- UPDATE, like READ, lets you read the object, but it also allows you update the object with SAVE.

Whether you specify READ or UPDATE, you can still make changes in the window, because the object does not change when you change the data in the window.

SIZE: The SIZE parameter specifies the address of the field where the system stores the size of the object. The system returns the size in this field whenever you specify SAVE or ACCESS with SIZE. If you omit SIZE or specify SIZE=*, the system does not return the size.

If you specified TYPE=DA with IDENTIFY for a data set object, SIZE specifies the address of a four-byte field. When control is returned to your program after the ACCESS service executes, the four-byte field contains the current size of the object. The size is the number of blocks that the application must map to ensure the mapping of the entire object.

If you specified TYPE=HS with IDENTIFY for a hiperspace object, ACCESS returns two sizes. The first is the current size of the hiperspace (in blocks). The second is the maximum size of the hiperspace (also in blocks). When specifying SIZE with an ID associated with a hiperspace object, you must provide an eight-byte field in which the system can return the sizes (4 bytes each).

LOCVIEW: The LOCVIEW parameter allows you to specify whether the system is to create a local copy of the data-in-virtual object.

If your object is a hiperspace, you cannot specify LOCVIEW=MAP.

If your object is a data set, you can code the LOCVIEW parameter two ways:

- LOCVIEW=MAP
- LOCVIEW=NONE (the default if you do not specify LOCVIEW)

If another program maps the same block of a data-in-virtual object as your program has mapped, a change in the object due to a SAVE by the other program can sometimes appear in the virtual storage window of your program. The change can appear when you allocate the data set object with a SHAREOPTIONS(2,3), SHAREOPTIONS(3,3), or SHAREOPTIONS(4,3) parameter, and when the other program is updating the object while your program is accessing it.

If the change appears when your program is processing the data in the window, processing results might be erroneous because the window data at the beginning of your processing is inconsistent with the window data at the end of your processing. For an explanation of SHAREOPTIONS, see *VSAM Administration Guide* or *Managing VSAM Data Sets*. The relationship between SHAREOPTIONS and LOCVIEW is as follows:

- When you allocate the data set object by SHAREOPTIONS(2,3), SHAREOPTIONS(3,3), or SHAREOPTIONS(4,3), the system does not serialize the accesses that programs make to the object. Under these options, if the programs do not observe any serialization protocol, the data in your virtual storage window can change when other programs invoke SAVE. To ensure that your program has a consistent view of the object, and protect your window from changes that other programs make on the object, use LOCVIEW=MAP. If you do not use LOCVIEW=MAP when you invoke ACCESS, the system provides a return code of 4 and a reason code of hexadecimal 37 as a reminder that no serialization is in effect even though the access was successful.
- When you allocate the object by SHAREOPTIONS(1,3), object changes made by the other program cannot appear in your window because the system performs automatic serialization of access. Thus, when any program has update access to the object, the system automatically prevents all other access. Use LOCVIEW=NONE when you allocate the data set by SHAREOPTIONS(1,3).

  Note: The usual method of programming data-in-virtual is to use LOCVIEW=NONE and SHAREOPTIONS(1,3). LOCVIEW=MAP is provided for programs that must access a data object simultaneously. Those programs would not use SHAREOPTIONS(1,3).

LOCVIEW=MAP requires extra processing that degrades performance. Use LOCVIEW=NONE whenever possible although you can use LOCVIEW=MAP for small data objects without significant performance loss. When you write a program that uses LOCVIEW=MAP, be careful about making changes in the object size. Consider the following:

- When a group of programs, all using LOCVIEW=MAP, have simultaneous access to the same object, no program should invoke any SAVE or MAP that extends or truncates the object unless it informs the other programs by some coding protocol of a change in object size. When the other programs are informed, they can adjust their processing based on the new size.
- All the programs must create their maps before any program changes the object size. Subsequently, if any program wants to reset the map or create a new map, it must not do so without observing the protocol of a size check. If the size changed, the program should invoke UNACCESS, followed by ACCESS to get the new size. Then the program can reset the map or create the new map.

The MAP Service

The MAP service makes an association between part or all of an object, the portion being specified by the OFFSET and SPAN parameters, and your program's virtual storage. This association, which is called a **virtual storage window,** takes the form of a one-to-one correspondence between specified blocks on the object and specified pages in virtual storage. After the MAP is complete, your program can then process the data in the window. The RETAIN parameter specifies whether data that is in the window when you issue MAP is to remain or be replaced by the data from the associated object.

Note: You cannot map virtual storage pages that are page-fixed into a virtual storage window. Once the window exists, you can page-fix data inside the window so long as it is not fixed when you issue SAVE, UNMAP, or RESET.

If your window is in an address space, you can map either a linear data set or a hiperspace object. See Figure 11-1.

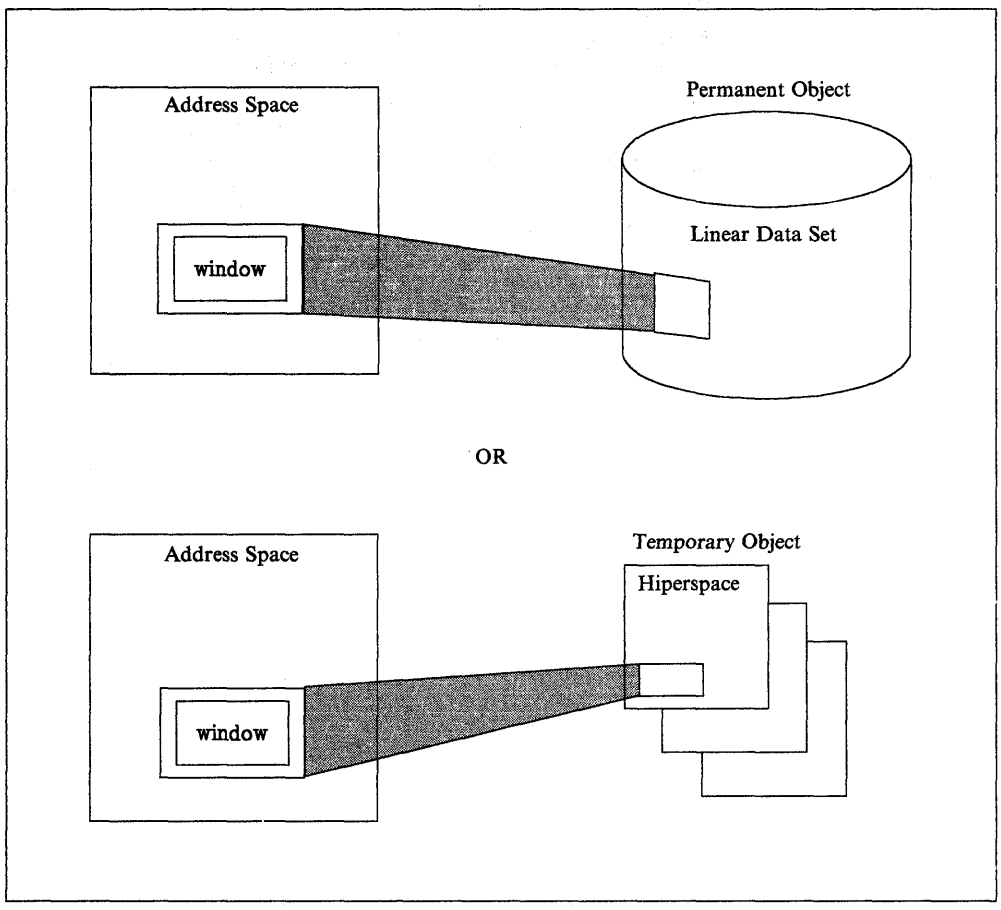


Figure 11-1. Mapping from an Address Space

If your window is in a data space or a hiperspace, you can map only a linear data set. See Figure 11-2.

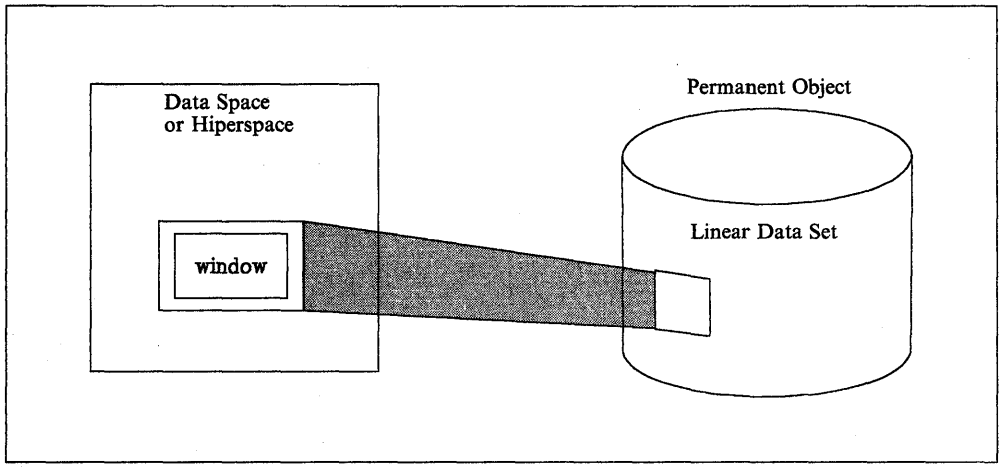


Figure 11-2. Mapping from a Data Space or Hiperspace

If your window is in a data space or hiperspace, you can issue multiple MAPs under the same ID to different data spaces or hiperspaces. You cannot mix data space

maps or hiperspace maps with address space maps under the same ID at any one time. However, you can mix data space maps and hiperspace maps. See Figure 11-3 on page 11-12.

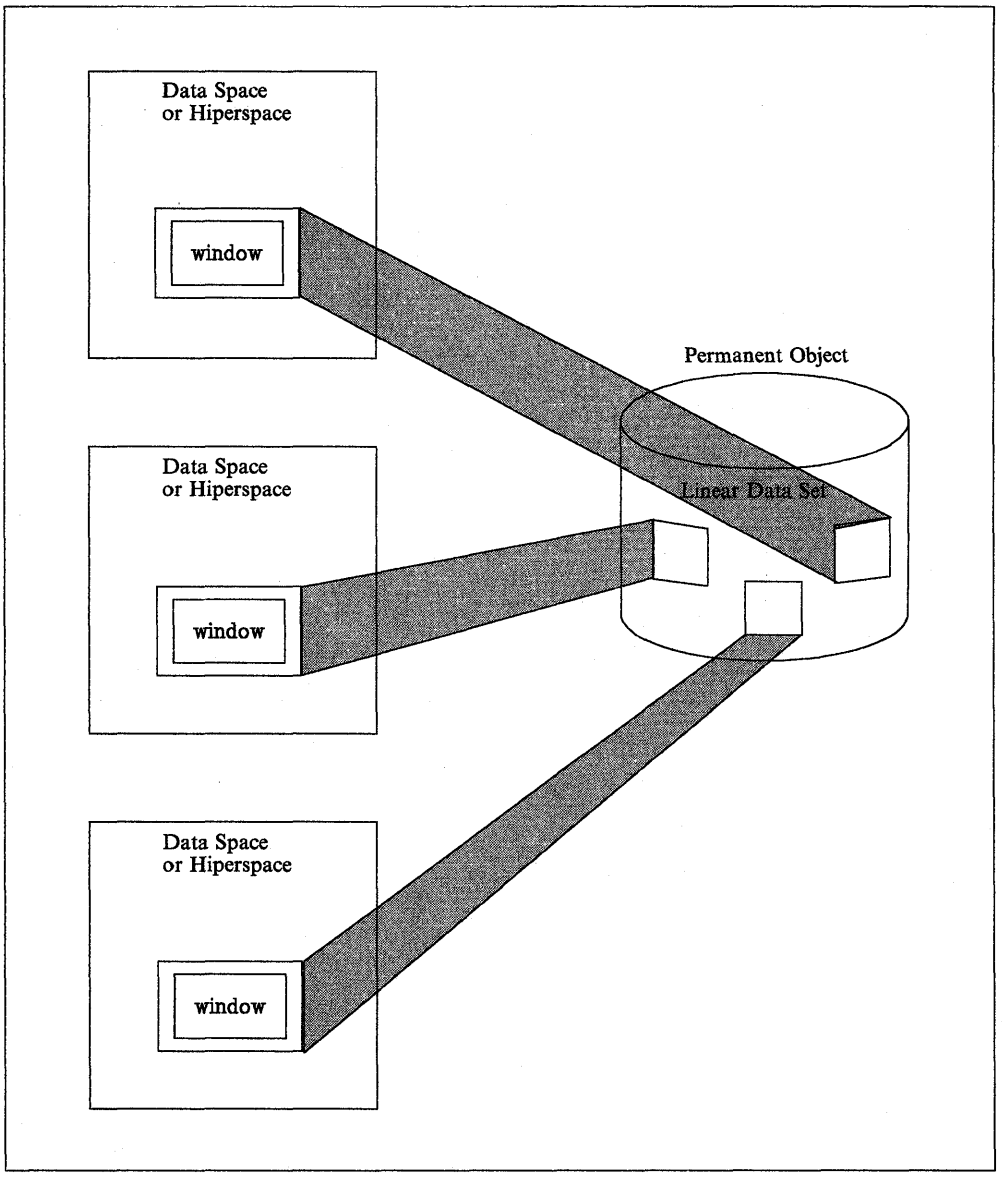


Figure 11-3. Multiple Mapping

The MAP service has seven parameters: ID, OFFSET, SPAN, AREA, RETAIN, STOKEN, and PFCOUNT.

The following examples show two ways to code the MAP service.

Hiperspace or data set object:

```
DIV MAP,ID=DIVOBJID,AREA=MAPPTR1,SPAN=SPANVAL,OFFSET=*,PFCOUNT=7
```

Data set object:

```
DIV MAP,ID=DIVOBJID,AREA=MAPPTR1,SPAN=SPANVAL,OFFSET=*,STOKEN=DSSTOK,PFCOUNT=7
```

ID: The ID parameter specifies the storage location containing the unique eight-byte value that was returned by IDENTIFY. The map service uses this value to determine which object is being mapped under which request.

If you specify the same ID on multiple invocations of the MAP service, you can create simultaneous windows corresponding to different parts of the object. However, an object block that is mapped into one window cannot be mapped into any other window created under the same ID. If you use different IDs, however, an object block can be included simultaneously in several windows.

OFFSET and SPAN: The OFFSET and SPAN parameters indicate a range of blocks on the object. Blocks in this range appear in the window. OFFSET indicates the first object block in the range, while SPAN indicates how many contiguous blocks make up the range. An offset of zero indicates the beginning of the object. For example, an offset of zero and a span of ten causes the block at the beginning of the object to appear in the window, together with the next nine object blocks. The window would then be ten pages long.

Specifying OFFSET=* or omitting OFFSET causes the system to use a default OFFSET of zero. Specifying SPAN=0, SPAN=*, or omitting SPAN results in a default SPAN of the number of blocks needed to MAP the entire object, starting from the block indicated by OFFSET. Specifying both OFFSET=* and SPAN=* or omitting both causes the entire object to appear in the window.

You may use the OFFSET and SPAN parameters to specify a range spanning any portion of the object, the entire object, or extending beyond the object. Specifying a range beyond the object enables a program to add data to the object, increasing the size of the object. If data in a mapped range beyond the object is saved (using the SAVE service), the size of the object is updated to reflect the new size.

To use the OFFSET parameter, specify the storage location containing the block offset of the first block to be mapped. The offset of the first block in the data object is zero. To use the SPAN parameter, specify the storage location containing the number of blocks in the mapped range.

AREA: When you specify MAP, you must also specify an AREA parameter. AREA indicates the beginning of a virtual storage space large enough to contain the entire window. Before invoking MAP, you must ensure that your task owns this virtual storage space. The storage must belong to a single, pageable private area subpool. It must begin on a 4096-byte boundary (that is, a page boundary) and have a length that is a multiple of 4096 bytes.

Note that any virtual storage space assigned to one window cannot be simultaneously assigned to another window. If your MAP request specifies a virtual storage location, via the AREA parameter, that is part of another window, the system rejects the request.

You cannot free virtual storage that is mapped into a window as long as the map exists. Attempts to do this will make the virtual space unusable and cause your program to abend. Subsequent attempts to reference the mapped virtual space will also cause an ABEND.

RETAIN: The RETAIN parameter determines what data you can view in the window. It can be either the contents of the virtual storage area (that corresponds to the window) the way it was before you invoked MAP, or it can be the contents of the object. The following table shows how using the RETAIN parameter with MAP affects the contents of the window.

RETAIN=	Window view
NO (default)	Contents of mapped object
YES	Contents of virtual storage

If you specify RETAIN=NO, or do not specify the RETAIN parameter at all (which defaults to RETAIN=NO), the contents of the object replace the contents of the virtual storage whenever your program references a page in the window. Virtual storage that corresponds to a range beyond the end of the object appears as binary zeroes when referenced. You can use RETAIN=NO to change some data and save it back to the object.

If you specify RETAIN=YES, the window retains the contents of virtual storage. The contents of the window are not replaced by data from the object. If you issue a subsequent SAVE, the data in the window *replaces* the data on the object. If the window extends beyond the object and your program has not referenced the pages in the extending part of the window, the system does not save the extending pages. However, if your program has referenced the extending pages, the system does save them on the object, extending the object so it can hold the additional data.

You can also use RETAIN=YES to reduce the size of (truncate) the object. If the part you want to truncate is mapped with RETAIN=YES and the window consists of freshly obtained storage, the data object size is reduced at SAVE time.

If you want to have zeroes written at the end of the object, the corresponding virtual storage must be explicitly set to zero prior to the SAVE.

STOKEN: To reference an entire linear data set through a single window, a program might require a considerable amount of virtual storage. In this case, the program can use a data space or hiperspace to contain the window. If you want the virtual storage window to be in a data space or hiperspace, specify STOKEN when you invoke MAP. When you specify STOKEN, you provide an eight-byte input parameter that identifies the data space or hiperspace, and that was returned from DSPSERV CREATE.

However, do not place the window in a data space or hiperspace under the following circumstances:

- If the data space is a disabled reference (DREF) data space.
- If the object is accessed with LOCVIEW=MAP.
- If the data space or hiperspace belongs to another task. However, if your program is in supervisor state or has a system storage key, it can use a data space or hiperspace that belongs to another task provided that the other task is in the same primary address space as your program.

PFCOUNT: PFCOUNT is useful for referencing sequential data. Because you get a page fault the first time you reference each page, preloading successive pages decreases the number of page faults.

The PFCOUNT parameter (*nnn*) is an unsigned decimal number up to 255. When an application references a mapped object, PFCOUNT tells the system that the program will be referencing this object in a sequential manner. PFCOUNT might improve performance because it asks the system to preload *nnn* pages, if possible. The system reads in *nnn* successive pages only to the end of the virtual range of the mapped area containing the originally referenced page, and only as resources are available.

You can use REFPAT INSTALL to define a reference pattern for the mapped area. In response to REFPAT, the system brings multiple pages into central storage when referenced. In this case, the PFCOUNT value you specify on DIV is not in effect as long as the reference pattern is in effect. When REFPAT REMOVE removes the definition of the reference pattern, the PFCOUNT you specify on DIV is again in effect. For information on the REFPAT macro, see "Defining the Reference Pattern (REFPAT)" on page 15-5.

The SAVE Service

The SAVE service writes changed pages from the window to the object if the changed pages are within the range to be saved. When you invoke SAVE, you specify a single and continuous range of blocks in the data-in-virtual object. Any virtual storage windows inside this range are eligible to participate in the SAVE.

For a SAVE request to be valid, the object must currently be accessed with MODE=UPDATE, under the same ID as the one specified on this SAVE request. Because you can map an object beyond its current end, the object might be extended when the SAVE completes if there are changed pages beyond the current end at the time of the ACCESS. On the other hand, the SAVE truncates the object if freshly obtained pages are being saved that are mapped in a range that extends to or beyond the end of the object **and** additional non-freshly obtained pages beyond the object area are not also being saved. In either case, the new object size is returned to your program if you specify the SIZE parameter.

When the system writes the pages from the window to the object, it clears (sets to zeroes) blocks in the object that are mapped to freshly obtained pages in the window if either one of the following conditions is true:

- There are subsequent pages in the range being saved that are not freshly obtained
- The blocks mapped to the freshly obtained pages are not at the end of the object. That is, they are imbedded in the object somewhere before the last block of the object. If the blocks mapped to freshly obtained pages do extend to the end of the object and no subsequent non-freshly obtained pages are being saved, then the object is truncated by that number of blocks.

If you specified RETAIN=YES with MAP, SAVE treats pages in the window that you have not previously saved as changed pages and will write them to the object.

Notes:

1. Do not specify SAVE for a storage range that contains DREF or page fixed storage.
2. If data to be saved has not changed since the last SAVE, no I/O will be performed. The performance advantages of using data-in-virtual are primarily because of the automatic elimination of unnecessary read and write I/O operations.
3. The range specified with SAVE can extend beyond the end of the object.
4. The system does not save pages of an object that is not mapped to any window.
5. The system does not save pages in a window that lies outside the specified range.

The following example shows how to code the SAVE service for a hiperspace or data set object.

```
DIV SAVE,ID=DIVOBJID,SPAN=SPAVAL,OFFSET=*,SIZE=OBJSIZE
```

ID: The ID parameter tells the SAVE service which data object the system is writing to under which request. Use ID to specify the storage location containing the unique eight-byte name that was returned by IDENTIFY. You must have previously accessed the object with MODE=UPDATE under the same ID as the one specified for SAVE.

OFFSET and SPAN: Use the OFFSET and SPAN parameters to select a continuous range of object blocks within which the SAVE service can operate. OFFSET indicates the first block and SPAN indicates the number of blocks in the range. As in the MAP service, the offset and span parameters refer to object blocks; they do not refer to pages in the window.

Specifying OFFSET=* or omitting OFFSET causes the system to use the default offset (zero). The zero offset does not omit or skip over any of the object blocks, and it causes the range to start right at the beginning of the object. Specifying SPAN=0, SPAN=*, or omitting SPAN gives you the default span. The default span includes the first object block after the part skipped by the offset, and it includes the entire succession of object blocks up to and including the object block that corresponds to the last page of the last window.

When SAVE executes, it examines each virtual storage window established for the object. In each window, it detects every page that corresponds to an object block in the selected range. Then, if the page has changed since the last SAVE, the system writes the page on the object. (If the page has not changed since the last SAVE, it is already identical to the corresponding object block and there is no need for to save it.) Although SAVE discriminates between blocks on the basis of whether they have changed, it has the effect of saving all window pages that lie in the selected range. Specifying both OFFSET=* and SPAN=* or omitting both causes the system to save all changed pages in the window without exceptions.

To use the OFFSET parameter, specify the storage location containing the block offset of the first block to be saved. The offset of the first block in the object is zero. To use the SPAN parameter, specify the storage location containing the number of blocks in the range to be saved.

SIZE: When you specify SIZE after the SAVE completes, the system returns the size of the data object in the virtual storage location specified by the SIZE parameter. If you omit SIZE or specify SIZE=*, the system does not return the size value. If TYPE=DA, invoking SAVE can change the size of the object. If TYPE=HS, invoking SAVE has no effect on the size of the object.

The RESET Service

At times during program processing, your program might have made changes to pages in the virtual storage window, and might no longer want to keep those changes. RESET, which is the opposite of SAVE, replaces data in the virtual storage window with data from the object. As with SAVE and MAP, the range to be reset refers to the object rather than the virtual storage. RESET resets only windows that are within the specified range, and it resets all the windows in the range that your program changed.

Do not specify RESET for a storage range that contains DREF storage.

Effect of RETAIN mode on RESET

You actually specify RETAIN on MAP, not on RESET, but the RETAIN mode of each individual window affects how the system resets the window. The following table shows the effect that issuing RETAIN with MAP has on RESET.

RETAIN=	RESET results
NO (default)	The data in the window matches the object data as of the last SAVE.
YES	Unless saved, the data in the window become freshly obtained. Any pages previously saved re-appear in their corresponding window. All other pages appear freshly obtained.

The system resets the window as follows:

- If you specified RETAIN=NO with MAP, after the RESET, the data in the window matches the object data as of the last SAVE. This applies to all the pages in the window.
- If you specified RETAIN=YES with MAP, the pages in the window acquire a freshly obtained status after the RESET unless you have been doing SAVE operations on this window. Individual object blocks changed by those SAVE operations re-appear after the RESET in their corresponding window pages, together with the other pages. However, the other pages appear freshly obtained.

Note: Regardless of the RETAIN mode of the window, any window page that corresponds to a block beyond the end of the object appears as a freshly obtained page.

The following example shows how to code the RESET service for a hiperspace or data set object:

```
DIV RESET,ID=DIVOBJID,SPAN=SPANVAL,OFFSET=*,RELEASE=YES
```

ID: The ID parameter tells the RESET service what data object is being written to. Use ID to specify the storage location containing the unique eight-byte name that was returned by IDENTIFY and used with previous MAP requests. You must have previously accessed the object (with either MODE=READ or MODE=UPDATE) under the same ID as the one currently specified for RESET.

OFFSET and SPAN: The OFFSET and SPAN parameters indicate the RESET range, the part of the object that is to supply the data for the RESET. As with MAP and SAVE, OFFSET indicates the first object block in the range, while SPAN indicates how many contiguous blocks make up the range, starting from the block indicated by OFFSET. The first block of the object has an offset of zero.

To use OFFSET, specify the storage location containing the block offset of the first block to be reset. To use SPAN, specify the storage location containing the number of blocks in the range to be RESET. Specifying OFFSET=* or omitting OFFSET causes the system to use a default OFFSET of zero. Specifying SPAN=* or omitting SPAN sets the default to the number of blocks needed to reset all the virtual storage windows that are mapped under the specified ID starting only with the block number indicated by OFFSET. Specifying both OFFSET=* and SPAN=* or omitting both resets all windows that are currently mapped under the specified ID.

RELEASE: RELEASE=YES tells the system to release all pages in the reset range. RELEASE=NO does not replace unchanged pages in the window with a new copy of pages from the object. It replaces only changed pages. Another ID might have changed the object itself while you viewed data in the window. Specify RELEASE=YES to reset all pages. Any subsequent reference to these pages causes the system to load a new copy of the data page from the object.

The UNMAP Service

Your program uses the UNMAP service to remove the association between a window in virtual storage and the object. Each UNMAP request must correspond to a previous MAP request. Note that UNMAP has no effect on the object. If you made changes in virtual storage but have not yet saved them, the system does not save them on the object when you issue UNMAP. UNMAP has four parameters: ID, AREA, RETAIN, and STOKEN.

The following examples show two ways to code the UNMAP service.

Hiperspace or data set object:

```
DIV UNMAP,ID=DIVOBJID,AREA=MAPPTR1
```

Data set object:

```
DIV UNMAP,ID=DIVOBJID,AREA=MAPPTR1,STOKEN=DSSTOK
```

ID: The ID parameter you specify is the address of an eight-byte field in storage. That field contains the identifier associated with the object. The identifier is the same value that the IDENTIFY service returned, which is also the same value you specified when you issued the corresponding MAP request.

AREA: The AREA parameter specifies the address of a four-byte field in storage that contains a pointer to the start of the virtual storage to be unmapped. This address must point to the beginning of a window. It is the same address that you provided when you issued the corresponding MAP request.

RETAIN: RETAIN specifies the state that virtual storage is to be left in after it is unmapped, that is, after you remove the correspondence between virtual storage and the object.

Specifying RETAIN=NO with UNMAP indicates that the data in the unmapped window is to be freshly obtained.

If your object is a hiperspace, you cannot specify RETAIN=YES. If your object is a data set, you can specify RETAIN=YES.

Specifying RETAIN=YES on the corresponding UNMAP transfers the data of the object into the unchanged pages in the window. In this case, RETAIN=YES with UNMAP specifies that the virtual storage area corresponding to the unmapped window is to contain the last view of the object. After UNMAP, your program can still reference and change the data in this virtual storage but can no longer save it on the object unless the virtual area is mapped again.

Notes:

1. If you issue UNMAP with RETAIN=NO, and there are unsaved changes in the virtual storage window, those changes are lost.
2. If you issue UNMAP with RETAIN=YES, and there are unsaved changes in the window, they remain in the virtual storage.
3. Unmapping with RETAIN=YES has certain performance implications. It causes the system to read unreferenced pages, and maybe some unchanged ones, from the object. You must not unmap with RETAIN=YES if your object is a hiperspace.
4. If the window is in a deleted data space, UNMAP works differently depending on whether you specify RETAIN=YES or RETAIN=NO. If you specify RETAIN=YES, the unmap fails and the program abends. Otherwise, the unmap is successful.

STOKEN: If you issued multiple maps under the same ID with different STOKENs, use STOKEN with UNMAP. If you do not specify STOKEN in this case, the system will scan the mapped ranges and unmap the first range that matches the specified virtual area regardless of the data space it is in. Issuing UNACCESS or UNIDENTIFY automatically unmaps all mapped ranges.

The UNACCESS and UNIDENTIFY Services

Use UNACCESS to terminate your access to the object for the specified ID. UNACCESS automatically includes an implied UNMAP. If you issue an UNACCESS with outstanding virtual storage windows, any windows that exist for the specified ID are unmapped with RETAIN=NO. The ID parameter is the sole parameter of the UNACCESS service, and it designates the same ID that you specified in the corresponding ACCESS. As in the other services, use ID to specify the storage location containing the unique eight-byte name that was returned by IDENTIFY.

Use UNIDENTIFY to notify the system that your use of an object under the specified ID has ended. If the object is still accessed as an object under this ID, UNIDENTIFY automatically includes an implied UNACCESS. The UNACCESS, in turn, issues any necessary UNMAPs using RETAIN=NO. The ID parameter is the only parameter for UNIDENTIFY, and it must designate the same ID as the one specified in the

corresponding ACCESS. As in the other services, use ID to specify the storage location containing the unique eight-byte name that was returned by IDENTIFY.

The following example shows how to code the UNACCESS and UNIDENTIFY services for a hiperspace or data set object:

```
DIV UNACCESS,ID=DIVOBJID
DIV UNIDENTIFY,ID=DIVOBJID
```

Sharing Data in an Object

When a user issues IDENTIFY, the system returns an ID and establishes an association between the ID and the user's task. All data-in-virtual services for a specific ID must be requested by the task that issued the IDENTIFY and obtained the ID.

Any task can reference or change the data in a mapped virtual storage window, even if the window was mapped by another task, and even if the object was identified and accessed by another task. Any task that has addressability to the window can reference or change the included data. However, only the task that issued the IDENTIFY can issue the SAVE to change the object.

When more than one user has the ability to change the data in a storage area, take the steps necessary to serialize the use of the shared area.

Miscellaneous Restrictions for Using Data-in-Virtual

- When you attach a new task, you cannot pass ownership of a mapped virtual storage window to the new task. That is, you cannot use the GSPV and GSPL parameters on ATTACH and ATTACHX to pass the mapped virtual storage.
- You cannot invoke data-in-virtual services in cross memory mode. There are no restrictions, however, against referencing and updating a mapped virtual storage window in cross memory mode.
- You cannot specify a non-shared standard hiperspace as a DIV object (DIV ACCESS) if you have issued ALESERV ADD for that hiperspace. You cannot issue ALESERV ADD for a non-shared standard hiperspace while it is a DIV object.

DIV Macro Programming Examples

The programming examples in this section illustrate how to code and execute a program that processes a data-in-virtual object. You can find additional examples, including illustrations, in:

- "Example of Mapping a Data-in-Virtual Object to a Data Space" on page 13-22
- "Using Data-in-Virtual with Hiperspaces" on page 13-37

General Program Description

This is a description of the program shown in "Data-in-Virtual Sample Program Code" on page 11-22.

Step 1. The program issues a DIV IDENTIFY and DIV ACCESS for the data-in-virtual object. The ACCESS returns the current size of the object in units of 4K bytes.

Step 2. If the object contains any data (the size returned by ACCESS is non-zero), the program issues a DIV MAP to associate the object with storage the program acquires using GETMAIN. The size of the MAP (and the acquired storage area) is the same as the size of the object.

Step 3. The program now processes the input statements from SYSIN. The processing depends upon the function requests (S, D, or E). If the program encounters an end-of-file, it treats it as if an "E" function was requested.

S function — Set a character in the object

Step 4. If the byte to change is past the end of the mapped area, the user asked to increase the size of the object. Therefore:

- Step a. If any changes have been made in the mapped virtual storage area but not saved to the object, the program issues a DIV SAVE. This save writes the changed 4K pages in the mapped storage to the object.
- Step b. The program issues a DIV UNMAP for the storage area acquired with GETMAIN, and then releases that area using FREEMAIN. The program skips this is step if the current object size is 0.
- Step c. The program acquires storage using GETMAIN to hold the increased size of the object, and issues a DIV MAP for this storage.

Step 5. The program changes the associated byte in the mapped storage. Note that this does not change the object. The program actually writes the changes to the object when you issue a DIV SAVE.

D function — Display a character in the object

Step 6. If the requested location is within the MAP size, the program references the specified offset into the storage area. If the data is not already in storage, a page fault occurs. Data-in-virtual processing brings the required 4K block from the object into storage. Then the storage reference is re-executed. The contents of the virtual storage area (i.e. the contents of the object) are displayed.

Step 7. **E function — End the program**

Step 8. If the program has made any changes in the mapped virtual storage area but has not saved them to the object, the program issues a DIV SAVE.

Step 9. The program issues a DIV UNIDENTIFY to terminate usage of the object. Note that data-in-virtual processing internally generates a DIV UNMAP and DIV UNACCESS.

Step 10. The program terminates.

Data-in-Virtual Sample Program Code

The first part of DIVSAMPL identifies the linear data set and accesses the object. If the object is not empty, the program obtains the virtual storage required to view (MAP) the entire object. Then it opens the input and message sequential data sets.

```
DIV      TITLE 'Data-in-Virtual Sample Program'
DIVSAMP  CSECT ,
DIVSAMP  AMODE 31                  Program runs in 31-bit mode
DIVSAMP  RMODE 24                  Program resides in 24-bit storage
         SAVE  (14,12),,'DIVSAMP -- Sample Program'
         LR    R11,R15             Establish base register
         USING DIVSAMP,R11         *
         LA    R2,VSVEAREA         Chain save areas together
         ST    R13,4(,R2)          *
         ST    R2,8(,R13)          *
         LR    R13,R2              *
* IDENTIFY and ACCESS the object pointed to by DD 'DIVDD'.
* Save the object's token in VTOKEN, and its size in VSIZEP.
         DIV   IDENTIFY,TYPE=DA,ID=VTOKEN,DDNAME=CDIVDD Specify DDNAME
         LA    R2,1                Error code
         LTR   R15,R15             IDENTIFY work ok ?
         BNZ   LERROR              * No -- quit
         DIV   ACCESS,ID=VTOKEN,MODE=UPDATE,SIZE=VSIZEP Open the object
         LA    R2,2                Error code
         LTR   R15,R15             ACCESS work ok ?
         BNZ   LERROR              * No -- quit
* If object not empty (VSIZEP > 0), get workarea to hold the object,
* and issue a MAP to it. The area must start on page boundary.
* Referencing byte "n" of this workarea gets byte "n" of the object.
         L     R2,VSIZEP           Current size (in 4K blocks)
         SLA   R2,12               Current size (in bytes)
         ST    R2,VSIZEB           VSIZEB = object size in bytes
         BZ    LEMPTY              If object not empty, get MAP area =
         GETMAIN RU,LV=(R2),LOC=(ANY,ANY),BNDRY=PAGE object size
         ST    R1,VAREAPTR         Save MAP area
         DIV   MAP,ID=VTOKEN,AREA=VAREAPTR,SPAN=VSIZEP
         LA    R2,3                Error code
         LTR   R15,R15             MAP work ok ?
         BNZ   LERROR              * No -- quit
LEMPTY   EQU   *                   Mapped, unless empty
* OPEN the SYSIN input data set, and SYSPRINT listing data set.
* Must be in 24-bit mode for this.  Then return to 31-bit mode.
         LA    R4,L31B01           Return to L31B01 in 31-bit mode
         LA    R1,L24B01           Go to L24B01 in 24-bit mode
         BSM   R4,R1               R4 = A(X'80000000'+L31B01)
L24B01   OPEN  (VSYSIN,(INPUT),VSYSPRT,(OUTPUT)) OPEN SYSIN/SYSPRINT
         BSM   0,R4                Return to 31-bit mode at next instr
L31B01   LA    R2,4                Error code from SYSIN OPEN
         LTR   R15,R15             OPEN ok ?
         BNZ   LERROR              * No -- quit
```

Data-in-Virtual Sample Program Code (continued)

The program reads statements from SYSIN until it reaches end-of-file, or encounters a statement with an "E" in column 1. The program validates the location in the object to set or display, and branches to the appropriate routine to process the request.

```
*
* Loop reading from SYSIN. Process the statements.
* Treat EOF as if the user specified "E" as the function to perform.
*
LREAD    EQU   *                  Read first/next card
         MVI   VCARDF,C'E'        EOF will appear as "E" function
         LA    R4,L31B02          Return to L31B02 in 31-bit mode
         LA    R1,L24B02          Go to L24B02 in 24-bit mode
         BSM   R4,R1              R4 = A(X'80000000'+L31B02)
L24B02   GET   VSYSIN,VCARD       Get the next input request.
LEOF     EQU   *                  End-of-file branches here
         BSM   0,R4               Return to 31-bit mode at next instr
L31B02   EQU   *                  Get here in 31-bit mode
*
* Process request:
* E                     - End processing
* S aaaaaaaa v          - Set location X'aaaaaaaa' to v
* D aaaaaaaa            - Display location X'aaaaaaaa'
*
         CLI   VCARDF,C'E'        EOF function or EOF on data set ?
         BE    LCLOSE             * Yes -- go cleanup and terminate
         TRT   VCARDA,CTABTRT     Ensure A-F, 0-9
         BNZ   LINVADDV           * If not, is error
         MVC   VTEMP8,VCARDA      Save address
         TR    VTEMP8,CTABTR      Convert to X'0A'-X'0F', X'00'-X'09'
         PACK  VCHGADDR(5),VTEMP8(9) Make address
         L     R1,VCHGADDR        Address
         LA    R1,0(,R1)          Clear hi-bit
         ST    R1,VCHGADDR        Save address to change/display
         CLI   VCARDF,C'D'        Display requested ?
         BE    LDISP              * Yes -- go process
         CLI   VCARDF,C'S'        Set requested ?
         BNE   LINVFUNC           * No -- is invalid statement
```

Data-in-Virtual Sample Program Code (continued)

For a set request, the program determines whether the location to change does not extend past the maximum object size allowed. If the location is past the end of the current window, the program saves any existing changes to the object, and creates a window containing the page to be changed. It then changes the data in storage (but not in the linear data set).

For a display request, the program ensures the location to display is in the linear object (that is, within the mapped area).

```
* SET: See if the location to change is within the range of the current
* MAP. If not, save any changes, get a larger area and issue a new MAP.
         C     R1,VSIZEB           Area to change within current MAP?
         BL    LGUPDCHR            * Yes -- continue
         C     R1,CSIZEMX          Area to change within max allowed?
         BNL   LINVADDR            * No -- is error
         CLI   VSWUPDT,0           Any updates to current MAP ?
         BE    LNOSVE1             * Yes -- then
         DIV   SAVE,ID=VTOKEN      Save any changes
         LA    R2,5                Error code from SAVE
         LTR   R15,R15             SAVE ok ?
         BNZ   LERROR              * No -- quit
         MVI   VSWUPDT,0           Clear update flag
LNOSVE1  L     R3,VSIZEB           Eliminate old map and storage
         LTR   R3,R3               Any to free ?
         BZ    LNOFREE             * Yes -- then
         DIV   UNMAP,ID=VTOKEN,AREA=VAREAPTR Release the MAP
         LA    R2,6                Error code from UNMAP
         LTR   R15,R15             UNMAP ok ?
         BNZ   LERROR              * No -- quit
         L     R1,VAREAPTR         R1 -> acquired storage
         FREEMAIN RU,A=(1),LV=(R3) Free the storage
LNOFREE  L     R2,VCHGADDR         Address of byte to change
         SRL   R2,12               R2 = page number - 1
         LA    R2,1(,R2)           R2 = page number to use
         ST    R2,VSIZEP           VSIZEP = MAP area in 4K units
         SLL   R2,12               R2 = size in bytes
         ST    R2,VSIZEB           VSIZEB = MAP area in bytes
         GETMAIN RU,LV=(R2),LOC=(ANY,ANY),BNDRY=PAGE get MAP area
         ST    R1,VAREAPTR         Save MAP area
         DIV   MAP,ID=VTOKEN,AREA=VAREAPTR,SPAN=VSIZEP
         LA    R2,3                Error code
         LTR   R15,R15             MAP work ok ?
         BNZ   LERROR              * No -- quit
LGUPDCHR L     R1,VCHGADDR         R1 =  byte to change
         A     R1,VAREAPTR         R1 -> byte to change
         MVC   0(1,R1),VCARDV      Change the byte
         MVI   VSWUPDT,X'FF'       Show change made
         B     LGOODINP            Go print accept message
LDISP    EQU   *                   Display location contents
         L     R1,VCHGADDR         R1 =  location to display
         C     R1,VSIZEB           Ensure within current MAP
         BNL   LINVADDR            * If not, is error
         A     R1,VAREAPTR         R1 -> location to display
         MVC   VCARDV,0(R1)        Put into the card
```

Data-in-Virtual Sample Program Code (continued)

For both the set and display requests, the program displays the character at the specified location. For an invalid request, the program displays an error message. For all requests, the program then goes to process another statement.

When requested to terminate, the program saves any changes in the linear data set, terminates its use of the object (using UNIDENTIFY), and returns to the operating system.

```
LGOODINP EQU   *
         MVC   M1A,VCARDA          Address changed/displayed
         MVC   M1B,VCARDV          Storage value
         CLI   M1B,X'00'           If X'00' (untouched),
         BNE   LGOODIN1            * change to "?".
         MVI   M1B,C'?'            *
LGOODIN1 LA    R2,M1               R2 -> message to print
         B     LTELL               Go tell user status
LINVFUNC LA    R2,M2               Unknown function
         B     LTELL               Go tell user status
LINVADDV LA    R2,M3               Invalid address
         B     LTELL               Go tell user status
LINVADDR LA    R2,M4               Address out of range
LTELL    EQU   *                   R2 -> message to print
         LA    R4,L31B03           Return to L31B03 in 31-bit mode
         LA    R1,L24B03           Go to L24B03 in 24-bit mode
         BSM   R4,R1               R4 = A(X'80000000'+L31B03)
L24B03   PUT   VSYSPRT,(R2)        Print the message
         BSM   0,R4                Return to 31-bit mode at next instr
L31B03   B     LREAD               Continue
* End-of-file on SYSIN, or "E" function requested.
* Save any changes (DIV SAVE). Then issue UNIDENTIFY, which internally
* issues UNMAP and UNIDENTIFY.
LCLOSE   EQU   *
         CLI   VSWUPDT,0           Any updates outstanding ?
         BE    LCLOSE1             * No -- skip SAVE
         DIV   SAVE,ID=VTOKEN      Save any changes
         LA    R2,5                Error code from SAVE
         LTR   R15,R15             SAVE ok ?
         BNZ   LERROR              * No -- quit
LCLOSE1  DIV   UNIDENTIFY,ID=VTOKEN All done with object
         LA    R2,6                Error code from UNIDENTIFY
         LTR   R15,R15             UNIDENTIFY ok ?
         BNZ   LERROR              * No -- quit
         L     R13,4(,R13)         Unchain save areas and return
         LM    R14,R12,12(R13)     *
         SR    R15,R15             *
         BR    R14                 *
LERROR   ABEND (R2),DUMP           Take a dump
```

Data-in-Virtual Sample Program Code (continued)

These are the program's variables.

```
* Variables and constants for the program
VSVEAREA DC    18A(0)              Save area
VTOKEN   DC    XL8'00'             Object token
VAREAPTR DC    A(*-*)              -> MAP area
VSIZEP   DC    F'0'                Size of MAP area, in pages (4K)
VSIZEB   DC    F'0'                Size of MAP area, in bytes
VSWUPDT  DC    X'00'               X'FF' -> map area updated
VCHGADDR DC    A(*-*),C' '         Address of byte to change/display
VTEMP8   DC    CL8' ',C' '         Temp area with buffer
VCARD    DC    CL80' '             Input card
VCARDF   EQU   VCARD+0,1           + Function (E/S/D)
VCARDA   EQU   VCARD+2,8           + Address to change/display
VCARDV   EQU   VCARD+11,1          + Character to change
CDIVDD   DC    X'5',C'DIVDD'       Linear Data Set DD pointer
* CTABTRT to verify string only has A thru F and 0 thru 9 (hex chars)
CTABTRT  DC    (C'A')X'FF',6X'00',(C'0'-C'F'-1)X'FF',10X'00',6X'FF'
* CTABTR & next line convert chars A:F,0:9 -> X'0A0B...0F000102...09'
CTABTR   EQU   *-C'A'
         DC    X'0A0B0C0D0E0F',(C'0'-C'F')X'00',X'010203040506070809'
CSIZEMX  DC    A(4096*1000)        Max size allowed for the DIV object
M1       DC    Y(M1E-*,0),C' Location '
M1A      DC    CL8' ',C' contains: '
M1B      DC    C' '
M1E      EQU   *
M2       DC    Y(M2E-*,0),C' Unknown function (not E/S/D)'
M2E      EQU   *
M3       DC    Y(M3E-*,0),C' Address not 8 hex characters'
M3E      EQU   *
M4       DC    Y(M4E-*,0),C' Address too big to set or display'
M4E      EQU   *
VSYSIN   DCB   MACRF=GM,DSORG=PS,RECFM=FB,LRECL=80,DDNAME=SYSIN,       *
               EODAD=LEOF
VSYSPRT  DCB   MACRF=PM,DSORG=PS,RECFM=VA,LRECL=133,DDNAME=SYSPRINT
R0       EQU   0                   Registers
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
         END   ,
```

Executing the Program

The following JCL executes the program called DIVSAMPL. The function of DIVSAMPL is to change and display bytes (characters) in the data-in-virtual object, DIV.SAMPLE, that was allocated in "Creating a Linear Data Set" on page 11-3.

```
//DIV JOB .....
//DIV  EXEC PGM=DIVSAMPL
//STEPLIB DD DISP=SHR,DSN=DIV.LOAD
//DIVDD   DD DISP=OLD,DSN=DIV.SAMPLE
//SYSABEND DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSIN DD *
S 00001000 A  Changes byte X'1000' to "A"
D 00000F00    Displays "?" since byte X'F00' contains X'00'
S 00000F00 B  Changes byte X'F00' to "B"
S 00010000 C  Saves previous changes, gets new map, changes byte X'10000'
D 00001000    Displays "A" which was set by first statement
D 00000F00    Displays "B" which was set by third statement
E             Saves changes since last save (stmt 4), and terminates pgm
/*
```

DIVSAMPL reads statements from SYSIN that tell the program what to do. The format of each statement is **f aaaaaaaa v**, where:

f Function to perform:
- **S** Set a character in the object.
- **D** Display a character in the object.
- **E** End the program.

aaaaaaaa The hexadecimal address of the storage to set or display. Leading 0s are required. The value must be less than X'003E8000'.

v For Set, the character to put into the object.

Note: The program actually saves the change requested by the S function when either the user asks to change a byte past the current size of the object, or the user asks to terminate the program (E function).

Chapter 12. Using Access Registers

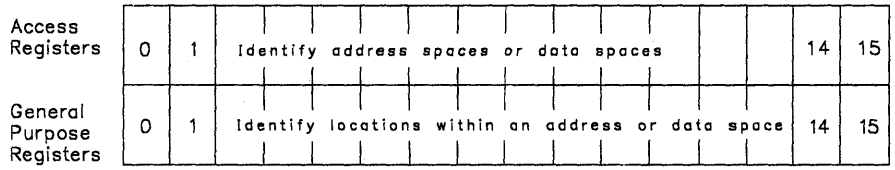
For storing data, MVS offers a program the use of a virtual storage area called a data space. assembler instructions (such as Load, Store, Add, and Move Character) manipulate the data in a data space. When you use instructions to manipulate data in a data space, your program must use the set of general purpose registers (GPRs) **plus** another set of registers called access registers. This chapter describes how to use access registers to manipulate data in data spaces.

Through access registers, your program can use assembler instructions to perform basic data manipulation, such as:

- Moving data into and out of a data space, and within a data space
- Performing arithmetic operations with values that are located in data spaces

To fully understand how to use the macros and instructions that control data spaces and access registers, you must first understand some concepts.

What is an access register (AR)? An AR is a hardware register that a program uses to identify an address space or a data space. Each processor has 16 ARs, numbered 0 through 15, and they are paired one-to-one with the 16 GPRs. When your program uses ARs, it must be in the address space control mode called access register (AR) mode.



ARs are used when fetching and storing data, but they are not used when doing branches.

What is address space control (ASC) mode? The ASC mode controls where the system looks for the data that the program is manipulating. Two ASC modes are available for your use: primary mode and access register (AR) mode.

- **In primary mode**, your program can access data that resides in the program's primary address space. When it resolves the addresses in data-referencing instructions, the system does not use the contents of the ARs.
- **In AR mode**, your program can access data that resides in the address space or data space that the ARs indicate. For data-referencing instructions, the system uses the AR and the GPR together to locate an address in an address space or data space.

How does your program switch ASC mode? Use the SAC instruction to change ASC modes:

- SAC 512 sets the ASC mode to AR mode.
- SAC 0 sets the ASC mode to primary mode.

What does an AR contain? An AR contains a token, an access list entry token (ALET). An ALET is an index to an entry on the access list. An **access list** is a table

of entries, each one of which points to an address space, data space, or hiperspace to which a program has access.

Figure 12-1 shows an ALET in the AR and the access list entry that points to an address space or a data space. It also shows the address in the GPR that points to the data within the address/data space.

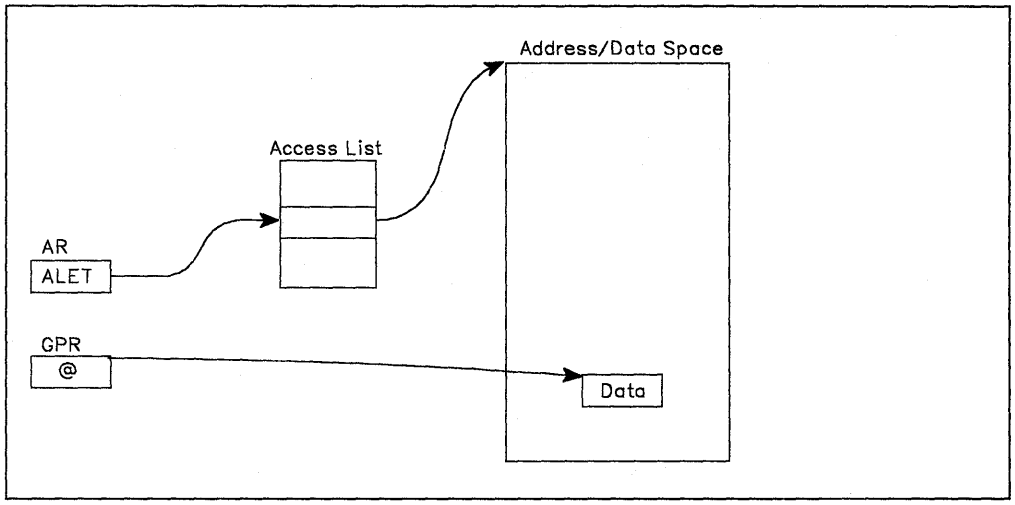


Figure 12-1. Using an ALET to Identify an an Address Space or a Data Space

For programs in AR mode, when the GPR is used as a base register in an instruction, the corresponding AR **must** contain an ALET. Conversely, when the GPR is not used as a base register, the corresponding AR is ignored.

By placing an entry on an access list and obtaining an ALET for the entry, a program builds the connection between the program and an address space, data space, or hiperspace. The process of building this connection is called **establishing addressability** to an address space, data space, or hiperspace. To add the entry to the access list, your program uses the ALESERV macro, which is described in "The ALESERV Macro" on page 12-9.

A program adds an entry to an access list so that it can:

- Gain access to **a data space or an address space** through assembler instructions.
- Obtain the ALET **for a hiperspace**. With that ALET, the program can use the HSPALET parameter on HSPSERV to:
  - Gain additional performance from the transfer of data to and from expanded storage. Information on when and how you use an access list entry for hiperspaces is described in "Obtaining Additional HSPSERV Performance" on page 13-31.
  - Improve its ability to share hiperspaces with other programs. The subject of sharing hiperspaces is described in "Shared and Non-shared Standard Hiperspaces" on page 13-28.

For the rest of this chapter, assume that entries in access lists point to data spaces, not hiperspaces or address spaces.

- The subject of inter-address space communication, appropriate only for programs in supervisor state or with PSW key 0 - 7, is described in *Extended Addressability Guide.*
- Because a program cannot use ARs to directly manipulate data in a hiperspace, the subject of how a program uses ARs and access lists to access hiperspaces differs from the discussion in the rest of this chapter.

Access Lists

When the system creates an address space, it gives that address space an access list (PASN-AL) that is empty. Programs add entries to the DU-AL and the PASN-AL. The entries represent the data spaces and hiperspaces that the programs want to access.

Types of Access Lists

An access list can be one of two types:

- A dispatchable unit access list (DU-AL), the access list that is associated with the TCB
- A primary address space access list (PASN-AL), the access list that is associated with the primary address space

Figure 12-2 on page 12-4 shows PGM1 that runs in AS1 under TCB A. The figure shows TCB A's DU-AL. It is available to PGM1 (and to other programs that TCB A might represent). The DU-AL has an entry for Data Space X, and PGM1 has the ALET for Data Space X. Therefore, PGM1 has access to Data Space X. PGM1 received an ALET for Space Y from another program. The PASN-AL has the entry for Space Y. Therefore, PGM1 also has access to Data Space Y. Because it does not have the ALET for Space Z, PGM1 cannot access data in Space Z.

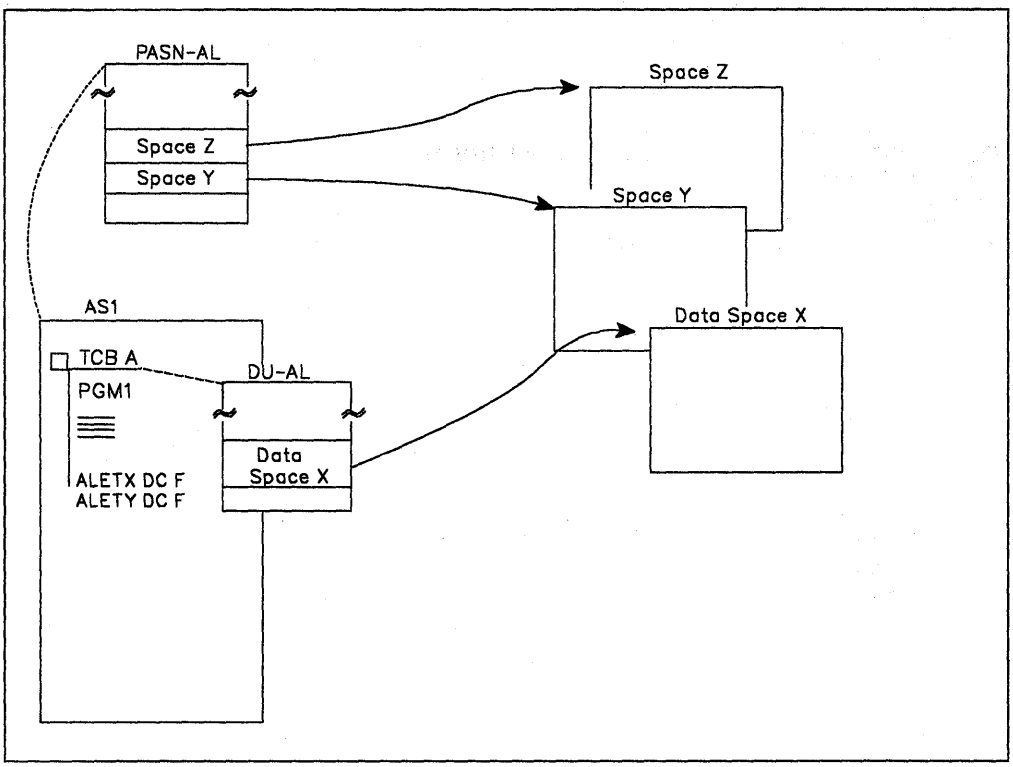


Figure 12-2. An Illustration of a DU-AL

The differences between a PASN-AL and a DU-AL are significant. The following table summarizes the characteristics of PASN-ALs and DU-ALs so you can understand how they differ.

Figure 12-3. Characteristics of DU-ALs and PASN-ALs

DU-AL	PASN-AL
Each work unit has its own unique DU-AL.	Every address space has its own unique PASN-AL. All programs running in the primary address space can access address spaces, data spaces, and hiperspaces through the PASN-AL.
All programs that the work unit represents can add and delete entries on the work unit's DU-AL.	Authorized programs (those in supervisor state programs or with PSW key 0 - 7) can add and delete entries on the PASN-AL. By passing an ALET to your program, these programs allow you to share the data in the spaces those ALETs represent.
A program cannot pass its task's DU-AL to a program running under another task. Tasks can never share a DU-AL. The one exception is that a program can pass a copy of its DU-AL to an attached task. This allows the new subtask to start with an identical copy of the attaching task's DU-AL. The two DU-ALs do not necessarily stay identical. After the ATTACH, the attaching task and the subtask are free to add and delete entries on their own DU-ALs. If the attaching task deletes the data space and the DU-AL entry for that data space, the subtask will still have an entry in its own DU-AL for that data space, but no program will be able to access this data space from the subtask.	All programs running with this address space as the primary address space can access address or data spaces through the PASN-AL.
A DU-AL can have up to 253 entries.	A PASN-AL can have up to 254 entries, some of which are reserved for the type of space called SCOPE=COMMON.
Your program can add entries to the DU-AL for the data spaces it created or owns.	Your problem state program with PSW key 8 - F can add entries for the data spaces it owns or created to the PASN-AL if an entry for the data space is not already on the PASN-AL. The data space entry may already be on the PASN-AL if another problem state program with PSW key 8 - F issued ALESERV ADD.

Writing Programs in AR Mode

After your program has an entry on an access list and the ALET that indexes the entry, it must place a value in an AR before it can use the data space. To understand how the system resolves addresses in instructions for programs in AR mode, see Figure 12-4. This figure shows how an MVC instruction in AR mode moves data from location B in one data space to location A in another data space:

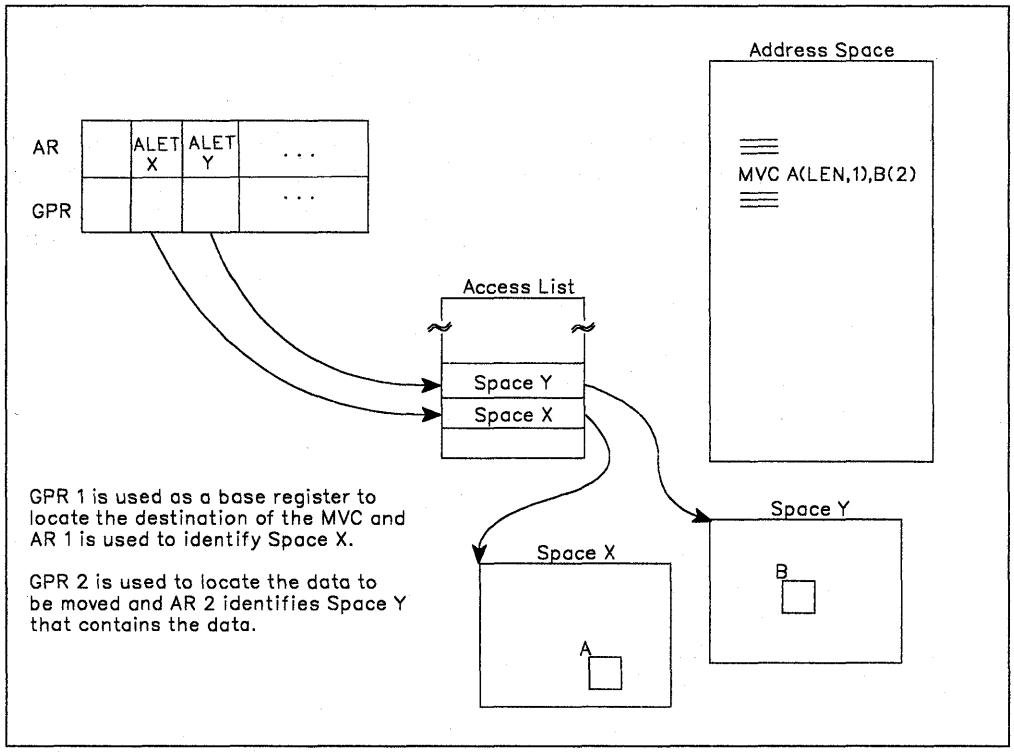


Figure 12-4. Using Instructions in AR Mode

GPR 1 is used as a base register to locate the destination of the data, and AR 1 is used to identify space X. GPR 2 is used to locate the source of the data, and AR 2 identifies Space Y. In AR mode, a program can use a single MVC instruction to move data from one address/data space to another. Note that the address space that contains the MVC instruction does not have to be either Space X or Space Y.

In similar ways, you can use instructions that compare, test-under-mask, copy, move, and perform arithmetic operations.

When the instructions reference data in the primary address space, the ALET in the AR must indicate that the data is in that address space. For this purpose, the system provides a special ALET with a value of zero. Other than using this value to identify the primary address space, a program should never depend on the value of an ALET.

> An ALET of zero designates the primary address space.

"Loading the Value of Zero into an AR" on page 12-8 shows several examples of loading a value of zero in an AR.

Rules for Coding Instructions in AR Mode

As you write your AR mode programs, use the advice and warnings in this section.

- For an instruction that references data, the system uses the contents of an AR to identify the address/data space that contains the data that the associated GPR points to.
- Use ARs only for data reference; do not use them with branching instructions.
- Just as you do not use GPR 0 as a base register, do not use AR/GPR 0 for addressing.
- An AR should contain only ALETs; do not store any other kinds of data in an AR.

Because ARs that are associated with index registers are ignored, **when you code assembler instructions in AR mode, place the commas very carefully.** In those instructions that use both a base register and an index register, the comma that separates the two values is very important. Figure 12-5 shows four examples of how a misplaced comma can change how the processor resolves addresses on the load instruction.

Figure 12-5. Base and Index Register Addressing in AR Mode

Instruction	Address Resolution
L 5,4(,3) or L 5,4(0,3)	There is no index register. GPR 3 is the base register. AR 3 indicates the address/data space.
L 5,4(3) or L 5,4(3,0)	GPR 3 is the index register. Because there is no base register, data is fetched from the primary address space.
L 5,4(6,8)	GPR 6 is the index register. GPR 8 is the base register. AR 8 indicates the address/data space.
L 5,4(8,6)	GPR 8 is the index register. GPR 6 is the base register. AR 6 indicates the address/data space.

For the first two entries in Figure 12-5:

In primary mode, the examples of the load instruction give the same result.

In AR mode, the data is fetched using different ARs. In the first entry, data is fetched from the address/data space represented by the ALET in AR 3. In the second entry, data is fetched from the primary address space (because AR/GPR 0 is not used as a base register).

For the last two entries in Figure 12-5:

In primary mode, the examples of the load instruction give the same result.

In AR mode, the first results in a fetch from the address/data space represented by AR 8, while the second results in a fetch from the address/data space represented by AR 6.

Manipulating the Contents of ARs

Whether the ASC mode of a program is primary or AR, it can use assembler instructions to save, restore, and modify the contents of the 16 ARs. The set of instructions that manipulate ARs include:

- CPYA — Copy the contents of an AR into another AR.
- EAR — Copy the contents of an AR into a GPR.
- LAE — Load a specified ALET and address into an AR/GPR pair.
- SAR — Place the contents of a GPR into an AR.
- LAM — Load the contents of one or more ARs from a specified storage location.
- STAM — Store the contents of one or more ARs to a specified storage location.

For their syntax and help with how to use these instruction, see *Principles of Operation.*

Loading an ALET into an AR

An action that is very important when a program is in AR mode, is the loading of an ALET into an AR. The following example shows how the LAM instruction loads an ALET into AR 2:

```
        LAM   2,2,DSALET          LOAD ALET OF DATA SPACE INTO AR2
*
DSALET  DS    F                   DATA SPACE ALET
```

Loading the Value of Zero into an AR

When the code you are writing is in AR mode, you must be very conscious of the contents of the ARs. For instructions that reference data, the ARs **must** always contain the ALET that identifies the data space that contains the data. When the data is in the primary address space, the AR that accompanies the GPR that has the address of the data must contain the value zero.

The following examples show several ways of placing the value zero in an AR.

Example 1: Set AR 5 to value of zero, when GPR 5 can be changed.

```
        SLR   5,5             SET GPR 5 TO ZERO
        SAR   5,5             LOAD GPR 5 INTO AR 5
```

Example 2: Set AR 5 to value of zero, without changing value in GPR 5.

```
        LAM   5,5,=F'0'       LOAD AR 5 WITH A VALUE OF ZERO
```

Another way of doing this is the following:

```
        LAM   5,5,ZERO
ZERO    DC    F'0'
```

Example 3: Set AR 5 to value of zero, when AR 12 is already zero.

```
        CPYA  5,12            COPY AR 12 INTO AR 5
```

Example 4: Set AR 12 to zero and set GPR 12 to the address contained in GPR 15. This sequence is useful to establish a program's base register GPR and AR from an entry point address contained in register 15.

```
PGMA     CSECT                ENTRY POINT
          .
          .
         LAE  12,0(15,0)      ESTABLISH PROGRAM'S BASE REGISTER
         USING PGMA,12
```

Another way to establish AR/GPR module addressability through register 12 is as follows:

```
         LAE   12,0
         BASR  12,0
         USING *,12
```

Example 5: Set AR 5 and GPR 5 to zero.

```
         LAE   5,0(0,0)       Set GPR and AR 5 to zero
```

The ALESERV Macro

Use the ALESERV macro to add an entry to an access list and delete that entry. The following sections describe the parameters on the ALESERV macro and give examples of its use.

Adding an Entry to an Access List

The ALESERV ADD macro adds an entry to the access list. Two parameters are required: STOKEN, an input parameter, and ALET, an output parameter.

- STOKEN – the eight-byte STOKEN of the address/data space represented by the entry. You might have received the STOKEN from DSPSERV or from another program.
- ALET — index to the entry that ALESERV added to the access list. The system returns this value at the address you specify on the ALET parameter.

The best way to describe how you add an entry to an access list is through an example. The following code adds an entry to a DU-AL. Assume that the DSPSERV macro has created the data space and has returned the STOKEN of the data space in DSPCSTKN and the origin of the data space in DSPCORG. ALESERV ADD returns the ALET in DSPCALET. The program then establishes addressability to the data space by loading the ALET into AR 2 and the origin of the data space into GPR 2.

```
*  ESTABLISH ADDRESSABILITY TO THE DATA SPACE
          .
          ALESERV ADD,STOKEN=DSPCSTKN,ALET=DSPCALET
          .
          LAM   2,2,DSPCALET          LOAD ALET OF SPACE INTO AR2
          L     2,DSPCORG             LOAD ORIGIN OF SPACE INTO GR2
          USING DSPCMAP,2             INFORM ASSEMBLER
          .
          L     5,DSPWRD1             GET FIRST WORD FROM DATA SPACE
                                      USES AR/GPR 2 TO MAKE THE REFERENCE
          .
DSPCSTKN DS   CL8                      DATA SPACE STOKEN
DSPCALET DS   F                        DATA SPACE ALET
DSPCORG  DS   F                        DATA SPACE ORIGIN RETURNED
DSPCMAP  DSECT                         DATA SPACE STORAGE MAPPING
DSPWRD1  DS    F                       WORD 1
DSPWRD2  DS    F                       WORD 2
DSPWRD3  DS    F                       WORD 3
```

Using the DSECT that the program established, the program can easily manipulate data in the data space.

It is possible to use ALESERV ADD to obtain an entry for a hiperspace. For information on how hiperspaces use ALETs, see "Obtaining Additional HSPSERV Performance" on page 13-31. Do not use ALESERV ADD for hiperspaces unless the move-page facility feature is installed on the processor.

Deleting an Entry from an Access List

Use ALESERV DELETE to delete an entry on an access list. The ALET parameter identifies the specific entry. It is a good programming practice to delete entries from an access list when the entries are no longer needed.

The following example deletes the entry that was added in the previous example.

```
          ALESERV DELETE,ALET=DSPCALET    REMOVE DS FROM AL
          DSPSERV DELETE,STOKEN=DSPCSTKN  DELETE THE DS
          .
DSPCSTKN DS   CL8                      DATA SPACE STOKEN
DSPCALET DS   F                        DATA SPACE ALET
```

If the program does not delete an entry, the entry remains on the access list until the work unit terminates. At that time, the system frees the access list entry.

Issuing MVS Macros in AR Mode

Many MVS macro services support callers in both primary and AR modes. When the caller is in AR mode, the macro service must generate larger parameter lists at assembly time. The increased size of the list reflects the addition of ALET-qualified addresses. At assembly time, a macro service that needs to know whether a caller is in AR mode checks the global bit that SYSSTATE ASCENV=AR sets. Therefore, it is good programming practice to issue SYSSTATE ASCENV=AR when a program changes to AR mode and issues macros while in that mode. Then, when the program returns to primary mode, issue SYSSTATE ASCENV=P to reset the global bit.

When your program is in AR mode, keep in mind these two facts:

- Before you use a macro in AR mode, check the description of the macro in *Assembler Programming Reference*. If the description of the macro does not specifically state that the macro supports callers in AR mode, use the SAC instruction to change the ASC mode and use the macro in primary mode.
- ARs 14 through 1 are volatile across all macro calls, whether the caller is in AR mode or primary mode. Don't count on the contents of these ARs being the same after the call as they were before.

Example of Using SYSSTATE

Consider that a program changes ASC mode from primary to AR mode and, while in AR mode, issues the LINKX and STORAGE macros. When it changes ASC mode, it should issue the following:

```
SAC       512
SYSSTATE ASCENV=AR
```

The LINKX macro generates different code and addresses, depending on the ASC mode of the caller. During the assembly of LINKX, the LINKX macro service checks the setting of the global bit. Because the global bit indicates that the caller is in AR mode, LINKX generates code and addresses that are appropriate for callers in AR mode.

The STORAGE macro generates the same code and addresses whether the caller is in AR mode or primary mode. Therefore, the STORAGE macro service does not check the global bit.

When the program changes back to primary mode, it should issue the following:

```
SAC       0
SYSSTATE ASCENV=P
```

Using X-Macros

Some macro services, such as LINK and LINKX, offer two macros, one for callers in primary mode and one for callers in either primary or AR mode. The names of the two macros are the same, except the macro that supports both primary and AR mode caller ends with an "X." This book refers to these macros as "X-macros." The rules for using all X-macros, except ESTAEX, are:

- Callers in primary mode can invoke either macro.

  Some parameters on the X-macros, however, are not valid for callers in primary mode. Some parameters on the non X-macros are not valid for callers in AR mode. Check the macro descriptions in *Assembler Programming Reference* for these exceptions.

- Callers in AR mode should issue the X-macro after issuing the SYSSTATE ASCENV=AR macro.

  If a caller in AR mode issues the non X-macro, the system substitutes the X-macro and issues a message during assembly that informs you of the substitution.

Use ESTAEX if your program is in AR mode. Otherwise, you can use either ESTAE or ESTAEX.

If your program issues macros while it is in AR mode, make sure the macros support AR mode callers and that SYSSTATE ASCENV=AR is coded.

If you rewrite programs and use the X-macro instead of the non X-macro, you must change both the list and execute forms of the macro. If you change only the execute form of the macro, the system will not generate the longer parameter list that the X-macro requires.

Note that an X-macro generates a larger parameter list than the corresponding non X-macro. A program using the X-macros must provide a larger parameter list than if it used the non X-macro.

Formatting and Displaying AR Information

The interactive problem control system (IPCS) can format and display AR data. Use the ARCHECK subcommand to:

- Display the contents of an AR
- Display the contents of an access list entry

See *IPCS Command Reference* for more information about the ARCHECK subcommand.

Chapter 13. Data Spaces and Hiperspaces

For storing data, MVS offers a program a choice of two kinds of virtual storage areas for data only: data spaces and hiperspaces. In making the decision whether to use a hiperspace or data space, you might have the following questions:

- Does my program need virtual storage outside the address space?
- Which kind of virtual storage is appropriate for my program?

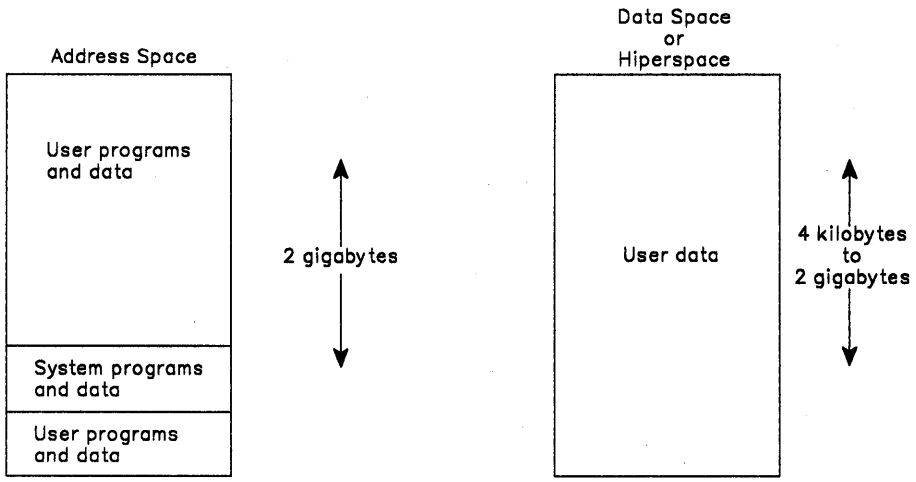
The first part of the chapter helps you make these decisions. Then, if you decide that one of these virtual storage areas would benefit your program, turn to one of the following sections for the information you need to create, use, and delete the area:

- "Creating and Using Data Spaces" on page 13-7
- "Creating and Using Hiperspaces" on page 13-26

What are Data Spaces and Hiperspaces?

Data spaces and hiperspaces are similar in that both are areas of virtual storage that the program can ask the system to create. The size of this space can range from four kilobytes to two gigabytes, according to the user's request. Unlike an address space, a data space or hiperspace contains only user data or user programs stored as data. Program code cannot run in a data space or a hiperspace.

The following diagram shows, at an overview level, the difference between an address space and a data space or hiperspace.



The major difference between a data space and a hiperspace is the way your program accesses data in the two areas. This difference is described later in this chapter. But before you can understand the differences, you need to understand what your program can do with these virtual storage areas.

What Can a Program Do With a Data Space or a Hiperspace?

Programs can use data spaces and hiperspaces to:

- Obtain more virtual storage than a single address space gives a user.
- Isolate data from other tasks in the address space.

  Data in an address space is accessible to all programs executing in that address space. You might want to move some data to a data space or hiperspace for security or integrity reasons. Use this space as a way to separate your data logically by its own particular use.
- Provide an area in which to map a data-in-virtual object.

You can place all types of data in a data space or hiperspace, rather than in an address space or on DASD. Examples of such data include:

- Tables, arrays, or matrixes
- Data base buffers
- Temporary work files
- Copies of permanent data sets

Because data spaces and hiperspaces do not include system areas, the cost of creating and deleting them is less than that of an address space.

To help you decide whether you need this additional storage area, some important questions are answered in the following sections. These same topics are addressed in greater detail later in the chapter.

How Does a Program Obtain a Data Space and a Hiperspace?

Data spaces and hiperspaces are created through the same system service: the DSPSERV macro. On this macro, you request either a data space or a hiperspace. You also specify some characteristics of the space, such as its size and its name.

The DSPSERV macro service gives you contiguous 31-bit addressable virtual storage of the size you specify and initializes the storage to binary zeroes.

Assembler Programming Reference contains the syntax and parameter descriptions for the macros that are mentioned in this chapter.

How Does a Program Move Data into a Data Space or Hiperspace?

One way to move data into a data space or a hiperspace is through buffers in the program's address space. Another way avoids using address space virtual storage as an intermediate buffer area: through data-in-virtual services, a program can move data into a data space or hiperspace directly. This second way reduces the amount of I/O.

Who Owns a Data Space or Hiperspace?

Although programs create data spaces and hiperspaces, they do not own them. When a program creates the space, the system assigns ownership to the TCB that represents the program, or to the TCB of the job step task of the program, if you choose. You can assign ownership of the data space to the job step TCB by specifying the TTOKEN option on the DSPSERV CREATE macro. All storage within a data space or hiperspace is available to programs that run under that TCB and, in some cases, the storage is available to other users. When the TCB terminates, the system deletes any data spaces or hiperspaces the TCB owns. If you want the data space to exist after the creating TCB terminates, assign the space to the job step

TCB. The job step will continue to be active beyond the termination of the creating TCB.

Because data spaces and hiperspaces belong to TCBs, keep in mind the relationship between the program and the TCB under which it runs. For simplicity, however, this chapter describes hiperspaces and data spaces as if they belong to programs. For example, "a program's data space" means "the data space that belongs to the TCB under which a program is running."

Can an Installation Limit the Use of Data Spaces and Hiperspaces?

The use of data spaces and hiperspaces consumes system resources such as expanded and auxiliary storage. Programmers responsible for tuning and maintaining MVS can set limits on the amount of virtual storage that programs in each address space can use for data spaces and hiperspaces. They can limit:

- The size of a single hiperspace or data space. (The default is 956K bytes, or 239 blocks.)
- The amount of storage available per address space for all hiperspaces and data spaces with a storage key of 8 - F. (The default is 2^{24} - 1 megabytes, or 16777215 megabytes.)
- The combined number of hiperspaces and data spaces with storage key 8 - F that can exist per address space at one time. (The default is 50 data spaces and hiperspaces.)

You should know the limits your installation establishes and the return codes that you can check to learn why the DSPSERV macro might not create the data space or hiperspace you requested.

How Does a Program Manage the Storage in a Data Space or Hiperspace?

Managing storage in data spaces or hiperspaces differs from managing storage in address spaces. Keep the following advisory notes in mind when you handle your data space storage:

- When you create a data space or hiperspace, use the DSPSERV macro to request a large enough size to handle your application.

  The amount of storage you specify when you create a data space or a hiperspace is the maximum amount the system will allow you to use in that space.

- You are responsible for keeping track of the allocating and freeing of data space and hiperspace storage. You cannot use the services of virtual storage management (VSM), such as the STORAGE, GETMAIN, or FREEMAIN macros, to manage this area. You can, however, use callable cell pool services to define a cell pool within a data space. You can then obtain the cells, as well as expand and contract the cell pool. "Using Callable Cell Pool Services to Manage Data Space Areas" on page 13-18 describes the use of callable cell pool services for data spaces. Information on how to code the services is in Chapter 10, "Callable Cell Pool Services."
- If you are not going to use an area of a data space or hiperspace again, release that area.
- When you are finished using a data space or hiperspace, delete it.

Differences Between Data Spaces and Hiperspaces

Up to this point, the chapter has focused on similarities between data spaces and hiperspaces. By now, you should know whether your program needs the kind of virtual storage that a data space or hiperspace offers. Only by understanding the differences between the two types of spaces, can you decide which one most appropriately meets your program's needs, or whether the program can use them both.

The main difference between data spaces and hiperspaces is the way a program references data. A program references data in a data space **directly**, in much the same way it references data in an address space. It addresses the data by the byte, manipulating, comparing, and performing arithmetic operations. The program uses the same instructions (such as load, compare, add, and move character) that it would use to access data in its own address space. To reference the data in a data space, the program must be in the ASC mode called access register (AR) mode. Pointers that associate the data space with the program must be in place and the contents of ARs that the instructions use must identify the specific data space.

Figure 13-1 shows a program in AR ASC mode using the data space. The CLC instruction compares data at two locations in the data space; the MVC instruction moves the data at location D in the data space to location C in the address space.

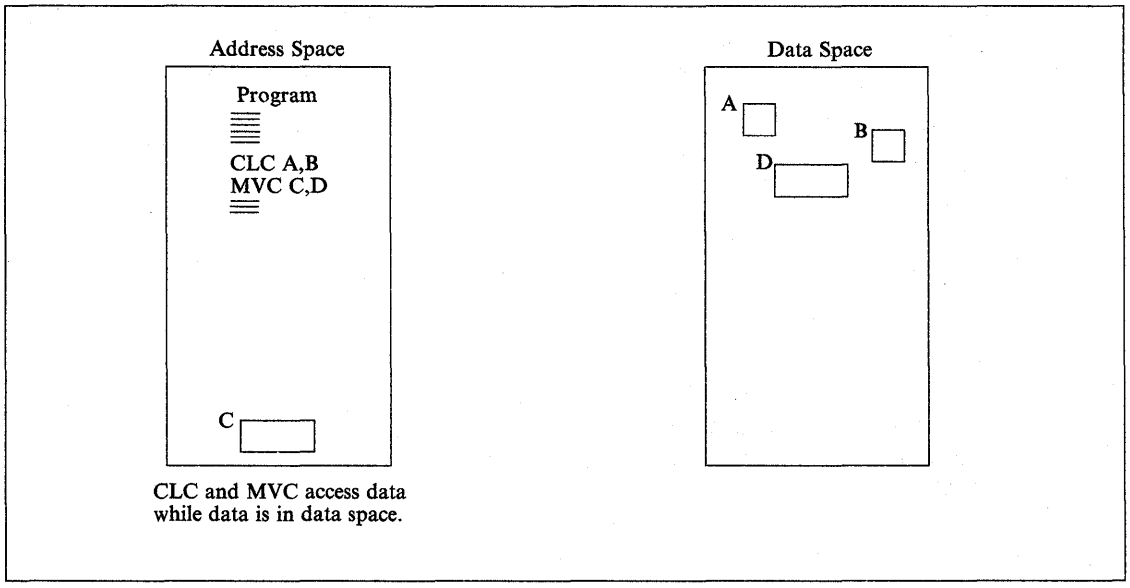


Figure 13-1. Accessing Data in a Data Space

In contrast, a program does not directly access data in a hiperspace. MVS provides a system service, the HSPSERV macro, to transfer the data between an address space and a hiperspace in 4K byte blocks. The HSPSERV macro read operation transfers the blocks of data from a hiperspace into an address space buffer where the program can manipulate the data. The HSPSERV write operation transfers the data from the address space buffer area to a hiperspace for storage. You can think of hiperspace storage as a high-speed buffer area where your program can store 4K byte blocks of data.

Figure 13-2 shows a program in an address space using the data in a hiperspace. The program uses the HSPSERV macro to transfer an area in the hiperspace to the address space, compares the values at locations A and B, and uses the MVC instruction to move data at location D to location C. After it finishes using the data in those blocks, the program transfers the area back to the hiperspace. The program could be in either primary or AR ASC mode.

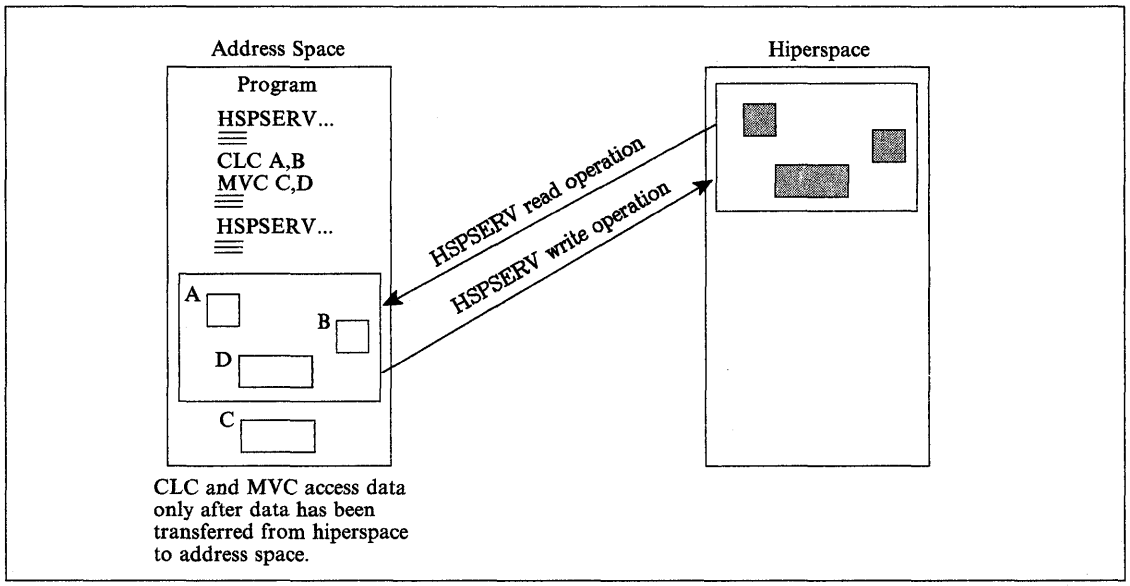


Figure 13-2. Accessing Data in a Hiperspace

On one HSPSERV macro, the program can transfer the data in more than one area between the hiperspace and the address space.

Comparing Data Space and Hiperspace Use of Physical Storage

To compare the performance of manipulating data in data spaces with the manipulating of data in hiperspaces, you should understand how the system "backs" these two virtual storage areas. (That is, what kind of physical storage the system uses to maintain the data in virtual storage.) The system uses the same resources to back data space virtual storage as it uses to back address space virtual storage: a combination of central and expanded storage (if available) frames, and auxiliary storage slots. The system can release low-use pages of data space storage to auxiliary storage and bring them in again when your program references those pages. The paging activity for a data space includes I/O between auxiliary storage paging devices and central storage.

The system backs hiperspace virtual storage with expanded storage, and auxiliary storage when expanded storage is not available. When you create a hiperspace, the system knows that the space will not be the target of assembler instructions and therefore will not need the backing of real frames. Therefore, data movement through HSPSERV does not include I/O activity between DASD and the expanded storage that backs the hiperspace pages. For this reason, hiperspaces are very efficient.

Which One Should Your Program Use?

If your program needs to manipulate or access data often by the byte, data spaces might be the answer. Use a data space if the program frequently addresses data at a byte level, such as you would in a workfile.

If your program can easily handle the data in 4K byte blocks, a hiperspace might give you the best performance. Use a hiperspace if the program needs a place to store data, but not to manipulate data. A hiperspace has other advantages:

- The program can stay in primary mode and ignore the access registers.
- The program can benefit from the high-speed access.
- The system can use the unused processor storage for other needs.

An Example of Using a Data Space

Suppose an existing program updates several rate tables that reside on DASD. Updates are random throughout the tables. The tables are too large and too many for your program to keep in contiguous storage in its address space. When the program updates a table, it reads that part of the table into a buffer area in the address space, updates the table, and writes the changes back to DASD. Each time it makes an update, it issues instructions that cause I/O operations.

If the tables were to reside in data spaces, one table to each data space, the tables would then be accessible to the program through assembler instructions. The program could move the tables to the data spaces (through buffers in the address space) once at the beginning of the update operations and then move them back (through buffers in the address space) at the end of the update operations.

If the tables are VSAM linear data sets, data-in-virtual can map the tables and move the data into the data space where a program can access the data. Data-in-virtual can then move the data from the data space to DASD. With data-in-virtual, the program does not have to move the data into the data space through address space buffers, nor does it have to move the data to DASD through address space buffers.

An Example of Using a Hiperspace

Suppose an existing program uses a data base that resides on DASD. The data base contains many records, each one containing personnel information about one employee. Access to the data base is random and programs reference but do not update the records. Each time the program wants to reference a record, it reads the record in from DASD.

If the data base were to exist in a hiperspace, the program would still bring one record into its address space at a time. Instead of reading from DASD, however, the program would bring in the records from the hiperspace on expanded storage (or auxiliary storage, when expanded storage is not available.) In effect, this technique can eliminate many I/O operations and reduce execution time.

Creating and Using Data Spaces

A **data space** is an area of virtual storage that a program can ask the system to create. Its size can range from 4K bytes to 2 gigabytes, according to the program's request. Unlike an address space, a data space contains only user data. Program code cannot run in a data space.

The DSPSERV macro manages data spaces. The TYPE=BASIC parameter (the default) tells the system that it is to manage a data space rather than a hiperspace. Use DSPSERV to:

- Create a data space
- Release an area in a data space
- Delete a data space
- Expand the amount of storage in a data space currently available to a program.
- Load an area of a data space into central storage
- Page an area of a data space out of central storage

Before it describes how your program can perform these actions, this chapter describes how your program will reference data in the data space it creates.

Manipulating Data in a Data Space

assembler instructions (such as load, store, add, and move character) manipulate the data in a data space. When you use instructions to manipulate data in a data space, your program must use the set of general purpose registers (GPRs) **plus** another set of registers called access registers. Chapter 12, "Using Access Registers" on page 12-1 describes how to use access registers to manipulate data in data spaces.

Rules for Creating, Deleting, and Managing Data Spaces

The SCOPE parameter determines what kind of data space a program creates. The three kinds of data spaces are SCOPE=SINGLE, SCOPE=ALL, and SCOPE=COMMON:

- SCOPE=SINGLE data spaces

  All programs can create, use, and delete SCOPE=SINGLE data spaces. Your program would use data spaces in much the same way as it uses private storage in an address space.

- SCOPE=ALL and SCOPE=COMMON data spaces

  Supervisor state or PSW key 0 - 7 programs can create, use, and delete data spaces that they can share with other programs. These data spaces have uses similar to MVS common storage.

To protect data in data spaces, the system places certain restrictions on problem state programs with PSW key 8 - F. The problem state programs with PSW key 8 - F can use SCOPE=ALL and SCOPE=COMMON data spaces, but they cannot create or delete them. They use them only under the control of supervisor state or PSW key 0 - 7 programs. This chapter assumes that the data spaces your program creates, uses, and deletes are SCOPE=SINGLE data spaces.

The following figure summarizes the rules for problem state programs with PSW key 8 - F:

Figure 13-3. Rules for How Problem State Programs with Key 8-F Can Use Data Spaces

Function	Rules
CREATE	Can create SCOPE=SINGLE data spaces.
DELETE	Can delete the data spaces it creates or owns, provided the PSW key of the program matches the storage key of the data space.
RELEASE	Can release storage in the data spaces it creates or owns, provided the PSW key of the program matches the storage key of the data space.
EXTEND	Can extend the current size of the data spaces it owns.
Add entries to the DU-AL	Can add entries to its DU-AL for the data spaces it created or owns.
Add entries to the PASN-AL	Can add entries to the PASN-AL for the data spaces it created or owns, providing an entry is not already on the PASN-AL as a result of an ALESERV ADD by a problem state program with PSW key 8 - F. If the ALET is already on the PASN-AL, the system does not create a duplicate entry, but the program can still access the data space using the ALET that already exists.
Access a data space through a DU-AL or PASN-AL	Can access a data space through its DU-AL and PASN-AL. The entry for a SCOPE=ALL or SCOPE=COMMON data space accessed through the PASN-AL must have been added to the PASN-AL by a program in supervisor state or PSW key 0 - 7. This program would have passed an ALET to the problem state PSW key 8 - F program.
LOAD	Can load an area of a data space it owns into central storage.
OUT	Can page an area of a data space it owns out from central storage.

There are other things that programs can do with data spaces. To do them, however, your program must be supervisor state or have a PSW key 0 - 7. For information on how these programs can use data spaces, see *Extended Addressability Guide*.

Creating a Data Space

To create a data space, issue the DSPSERV CREATE macro. MVS gives you contiguous 31-bit virtual storage of the size you specify and initializes the storage to hexadecimal zeroes.

On the DSPSERV macro, you are required to specify:

- The name of the data space (NAME parameter)

  To ask DSPSERV to generate a data space name unique to the address space, use the GENNAME parameter. DSPSERV will return the name it generates at the location you specify on the OUTNAME parameter. See "Choosing the Name of a Data Space" on page 13-9.

- A location where DSPSERV can return the STOKEN of the data space (STOKEN parameter)

  DSPSERV CREATE returns a STOKEN that you can use to identify the data space to other DSPSERV services and to the ALESERV and DIV macros.

Other information you might specify on the DSPSERV macro is:

- The maximum size of the data space and its initial size (BLOCKS parameter). If you do not code BLOCKS, the data space size is determined by defaults set by your installation. In this case, use the NUMBLKS parameter to tell the system where to return the size of the data space. See "Specifying the Size of a Data Space" on page 13-10.
- A location where DSPSERV can return the address (either 0 or 4096) of the first available block of the data space (ORIGIN parameter). See "Identifying the Origin of a Data Space" on page 13-11.
- The TTOKEN of the caller's job step task. If you want the data space to exist after your task terminates, or to be made concurrently available to any existing task in the job step as well as the creating task, assign ownership of the data space to the job step task. "Sharing Data Spaces among Problem State Programs with PSW Keys 8 - F" on page 13-20 describes a program that requests the TTOKEN of the job step task and then assigns ownership of a data space to the job step task. To request the TTOKEN of the job step task, issue the TCBTOKEN macro using the TYPE=JOBSTEP option.

Choosing the Name of a Data Space

The names of data spaces and hiperspaces must be unique within an address space. You have a choice of choosing the name yourself or asking the system to generate a unique name. To keep you from choosing names that it uses, MVS has some specific rules for you to follow. These rules are listed in the DSPSERV description under the NAME parameter in *Assembler Programming Reference*.

Use the GENNAME parameter to ask the system to generate a unique name. GENNAME=YES generates a unique name that has, as its last one to three characters, the first one to three characters of the name you specify on the NAME parameter.

Example 1: If PAYbbbbb is the name you supply on the NAME parameter and you code GENNAME=YES, the system generates the following name:

```
ncccPAY
```

where the system generates the digit *n* and the characters *cccc*, and appends the characters *PAY* that you supplied.

Example 2: If Jbbbbbbb is the name you supply on the NAME parameter and you code GENNAME=YES, the system generates the following name:

```
nccccJ
```

GENNAME=COND checks the name you supply on the NAME parameter. If it is already used for a data space or a hiperspace, DSPSERV supplies a name with the format described for the GENNAME=YES parameter.

To learn the unique name that the system generates for the data space or hiperspace you are creating, use the OUTNAME parameter.

Specifying the Size of a Data Space

When you create a data space or hiperspace, you tell the system on the BLOCKS parameter how large to make that space, the largest size being 524,288 blocks. (The product of 524288 times 4K bytes is 2 gigabytes.) The addressing range for the data space or hiperspace depends on the processor. If your processor does not support an origin of zero, the limit is actually 4096 bytes less than 2 gigabytes. Before you code BLOCKS, you should know two facts about how an installation controls the use of virtual storage for data spaces and hiperspaces.

- An installation can set limits on the amount of storage available for each address space for all data spaces and hiperspaces with a storage key of 8 through F. If your request for this kind of space (either on the DSPSERV CREATE or DSPSERV EXTEND) would cause the installation limit to be exceeded, the system rejects the request with a nonzero return code and a reason code.
- An installation sets a default size for data spaces and hiperspaces; you should know this size. If you do not use the BLOCKS parameter, the system creates a data space with the default size. (The IBM default size is 239 blocks.)

The data spaces and hiperspaces your programs create have a storage key greater than 7. The system adds the initial size of these spaces to the cumulative total of all data spaces and hiperspaces for the address space and checks this total against the installation limit. For information on the IBM defaults, see "Can an Installation Limit the Use of Data Spaces and Hiperspaces?" on page 13-3.

The BLOCKS parameter allows you to specify a **maximum size** and **initial size** value.

- The maximum size identifies the largest amount of storage you will need in the data space.
- An initial size identifies the amount of the storage you will immediately use.

As you need more space in the data space or hiperspace, you can use the DSPSERV EXTEND macro to increase the available storage. The amount of available storage is called the **current size.** (At the creation of a data space or hiperspace, the initial size is the same as the current size.) When it calculates the cumulative total of data space and hiperspace storage, the system uses the current size.

If you know the default size and want a data space or hiperspace smaller than or equal to that size, use the BLOCKS=maximum size or omit the BLOCKS parameter.

If you know what size data space or hiperspace you need and are not concerned about exceeding the installation limit, set the maximum size and the initial size the same. BLOCKS=0, the default, establishes a maximum size and initial size both set to the default size.

If you do not know how large a data space or hiperspace you will eventually need or you are concerned with exceeding the installation limit, set the maximum size to the largest size you might possibly use and the initial size to a smaller amount, the amount you currently need.

Use the NUMBLKS parameter to request that the system return the size of the space it creates for you. You would use NUMBLKS, for example, if you did not specify BLOCKS and do not know the default size.

Figure 13-4 on page 13-11 shows an example of using the BLOCKS parameter to request a data space with a maximum size of 100,000 bytes of space and a current size (or initial size) of 20,000 bytes. You would code the BLOCKS parameter on DSPSERV as follows:

```
         DSPSERV CREATE,. . .,BLOCKS=(DSPMAX,DSPINIT)
          .
DSPMAX  DC  A((100000+4095)/4096)          DATA SPACE MAXIMUM SIZE
DSPINIT DC  A((20000+4095)/4096)           DATA SPACE INITIAL SIZE
```

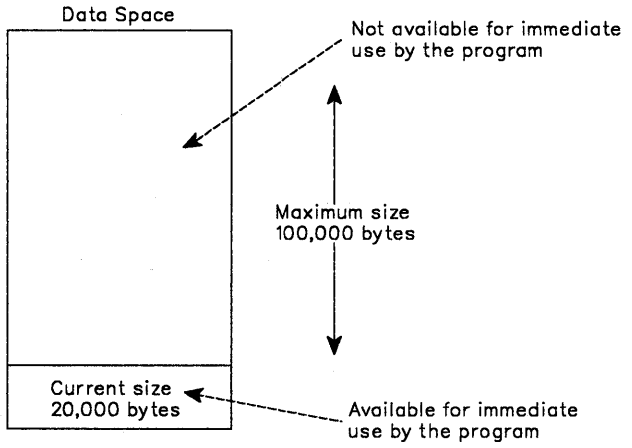


Figure 13-4. Example of Specifying the Size of a Data Space

As your program uses more of the data space storage, it can use DSPSERV EXTEND to extend the current size. "Extending the Current Size of a Data Space" on page 13-16 describes extending the current size and includes an example of how to extend the current size of the data space in Figure 13-4.

Identifying the Origin of a Data Space

Some processors do not allow the data space or hiperspace to start at zero; these spaces start at address 4096 bytes. When you use DSPSERV CREATE, you can count on the origin of the data space or hiperspace staying the same within the same IPL. To learn the starting address, either:

- Create a data space 1 block larger than you need and then assume that the space starts at address 4096, or
- Use the ORIGIN parameter.

If you use ORIGIN, the system returns the beginning address of the data space or hiperspace at the location you specify.

Unless you specify a size of 2 gigabytes and the processor does not support an origin of zero, the system gives you the size you request, regardless of the location of the origin.

An example of the problem you want to avoid in addressing data space storage is as follows:

> Suppose a program creates a data space of 1 megabyte and assumes the data space starts at address 0 when it really begins at the address 4096. Then, if the program uses an address lower than 4096 in the data space, the system abends the program.

Example of Creating a Data Space

In the following example, a program creates a data space named TEMP. The system returns the origin of the data space (either 0 or 4096) at location DSPCORG.

```
         DSPSERV CREATE,NAME=DSPCNAME,STOKEN=DSPCSTKN,              X
               BLOCKS=DSPBLCKS,ORIGIN=DSPCORG
         .
DSPCNAME DC   CL8'TEMP    '          DATA SPACE NAME
DSPCSTKN DS   CL8                    DATA SPACE STOKEN
DSPCORG  DS   F                      DATA SPACE ORIGIN RETURNED
DSPCSIZE EQU  10000000               10 MILLION BYTES OF SPACE
DSPBLCKS DC   A((DSPCSIZE+4095)/4096) NUMBER OF BLOCKS NEEDED FOR
*                                    A 10 MILLION BYTE DATA SPACE
```

Establishing Addressability to a Data Space

Creating a data space does not give you access to the data space. You must use the ALESERV macro and issue certain assembler instructions before you can use the data space. The ALESERV macro adds an entry to an access list, either the DU-AL or the PASN-AL. The STOKEN parameter identifies the data space and the ALET parameter tells ALESERV where to return the access list entry token (that is, the ALET).

Your program can add entries for the data spaces it created or owns to either the DU-AL or the PASN-AL. Programs that the work unit represents can use the DU-AL. All programs running in the primary address space can use the PASN-AL for that address space. If you want all programs in the address space to have access to the data space entries, your program should put the entries on the PASN-AL. If you want to restrict the use of the entries, your program should put the entries on the DU-AL. When you add an entry to the PASN-AL, however, the system checks to see if an entry for that data space already exists on the PASN-AL. If the ALET is already on the PASN-AL, the system does not create a duplicate entry, but the program can still access the data space.

When your program wants to manipulate data in the data space, it places the ALET in an AR and changes its ASC mode to AR mode. For examples of how to establish addressability to data spaces and manipulate data in those data spaces, see Chapter 12, "Using Access Registers." "The ALESERV Macro" on page 12-9 describes how to add access list entries and gives an example.

Examples of Moving Data into and out of a Data Space

When using data spaces, you sometimes have large amounts of data to transfer between the address space and the data space. This section contains examples of two subroutines, both named COPYDATA, that show you how to use the Move (MVC) and Move Long (MVCL) instructions to move a variable number of bytes into and out of a data space. (You can also use the examples to help you move data within an address space.) The two examples perform exactly the same function; both are included here to show you the relative coding effort required to use each instruction.

The use of registers for the two examples is as follows:

```
Input:  AR/GR 2     Target area location
        AR/GR 3     Source area location
        GR 4        Signed 32-bit length of area
                    (Note: A negative length is treated as zero.)
        GR 14       Return address
Output: AR/GR 2-14  Restored
        GR 15       Return code of zero
```

The routines can be called in either primary or AR mode; however, during the time they manipulate data in a data space, they must be in AR mode. The source and target locations are assumed to be the same length (that is, the target location is not filled with a padding character).

Example 1: Using the MVC Instruction: The first COPYDATA example uses the MVC instruction to move the specified data in groups of 256 bytes:

```
COPYDATA DS    0D
         BAKR  14,0                SAVE CALLER'S STATUS
         LAE   12,0(0,0)           BASE REG AR
         BALR  12,0                BASE REG GR
         USING *,12                ADDRESSABILITY
  .
         LTR   4,4                 IS LENGTH NEGATIVE OR ZERO?
         BNP   COPYDONE            YES, RETURN TO CALLER
  .
         S     4,=F'256'           SUBTRACT 256 FROM LENGTH
         BNP   COPYLAST            IF LENGTH NOW NEGATIVE OR ZERO
*                                  THEN GO COPY LAST PART
  .
COPYLOOP DS    0H
         MVC   0(256,2),0(3)       COPY 256 BYTES
         LA    2,256(,2)           ADD 256 TO TARGET ADDRESS
         LA    3,256(,3)           ADD 256 TO SOURCE ADDRESS
         S     4,=F'256'           SUBTRACT 256 FROM LENGTH
         BP    COPYLOOP            IF LENGTH STILL GREATER THAN
*                                  ZERO, THEN LOOP BACK
COPYLAST DS    0H
         LA    4,255(,4)           ADD 255 TO LENGTH
         EX    4,COPYINST          EXECUTE A MVC TO COPY THE
*                                  LAST PART OF THE DATA
         B     COPYDONE            BRANCH TO EXIT CODE
COPYINST MVC   0(0,2),0(3)         EXECUTED INSTRUCTION
COPYDONE DS    0H
  .
*  EXIT CODE
         LA    15,0                SET RETURN CODE OF 0
         PR                        RETURN TO CALLER
```

Example 2: Using the MVCL Instruction: The second COPYDATA example uses the MVCL instruction to move the specified data in groups of 1048576 bytes:

```
COPYDATA DS    0D
         BAKR  14,0                SAVE CALLER'S STATUS
         LAE   12,0(0,0)           BASE REG AR
         BALR  12,0                BASE REG GR
         USING *,12                ADDRESSABILITY
  .
         LA    6,0(,2)             COPY TARGET ADDRESS
         LA    7,0(,3)             COPY SOURCE ADDRESS
         LTR   8,4                 COPY AND TEST LENGTH
         BNP   COPYDONE            EXIT IF LENGTH NEGATIVE OR ZERO
  .
         LAE   4,0(0,3)            COPY SOURCE AR/GR
         L     9,COPYLEN           GET LENGTH FOR MVCL
         SR    8,9                 SUBTRACT LENGTH OF COPY
         BNP   COPYLAST            IF LENGTH NOW NEGATIVE OR ZERO
*                                   THEN GO COPY LAST PART
  .
COPYLOOP DS    0H
         LR    3,9                 GET TARGET LENGTH FOR MVCL
         LR    5,9                 GET SOURCE LENGTH FOR MVCL
         MVCL  2,4                 COPY DATA
         ALR   6,9                 ADD COPYLEN TO TARGET ADDRESS
         ALR   7,9                 ADD COPYLEN TO SOURCE ADDRESS
         LR    2,6                 COPY NEW TARGET ADDRESS
         LR    4,7                 COPY NEW SOURCE ADDRESS
         SR    8,9                 SUBTRACT COPYLEN FROM LENGTH
         BP    COPYLOOP            IF LENGTH STILL GREATER THAN
*                                   ZERO, THEN LOOP BACK
  .
COPYLAST DS    0H
         AR    8,9                 ADD COPYLEN
         LR    3,8                 COPY TARGET LENGTH FOR MVCL
         LR    5,8                 COPY SOURCE LENGTH FOR MVCL
         MVCL  2,4                 COPY LAST PART OF THE DATA
         B     COPYDONE            BRANCH TO EXIT CODE
COPYLEN  DC    F'1048576'          AMOUNT TO MOVE ON EACH MVCL
COPYDONE DS    0H
  .
*  EXIT CODE
         LA    15,0                SET RETURN CODE OF 0
         PR                        RETURN TO CALLER
```

Programming Notes for Example 2:

- The MVCL instruction uses GPRs 2, 3, 4, and 5.
- The ALR instruction uses GPRs 6, 7, 8, and 9.
- The maximum amount of data that one execution of the MVCL instruction can move is 2^{24}-1 bytes (16777215 bytes).

Extending the Current Size of a Data Space

When you create a data space and specify a maximum size larger than the initial size, you can use DSPSERV EXTEND to increase the current size as your program uses more storage in the data space. The BLOCKS parameter specifies the amount of storage you want to add to the current size of the data space.

The system increases the data space by the amount you specify, unless that amount would cause the system to exceed one of the following:

- The data space maximum size, as specified by the BLOCKS parameter on DSPSERV CREATE when the data space was created
- The installation limit for the combined total of data space and hiperspace storage with storage key 8 -F per address space. These limits are either the system default or are set in the installation exit IEFUSI.

If one of those limits would be exceeded, the VAR parameter tells the system how to satisfy the EXTEND request.

- VAR=YES (the variable request) tells the system to extend the data space as much as possible, without exceeding the limits set by the data space maximum size or the installation limits. In other words, the system extends the data space to one of the following sizes, depending on which is smaller:
  - The maximum size specified on the BLOCKS parameter
  - The largest size that would still keep the combined total of data space and hiperspace storage within the installation limit.
- VAR=NO (the default) tells the system to:
  - Abend the caller, if the extended size would exceed the maximum size specified at the creation of the data space
  - Reject the request, if the data space has storage key 8 - F and the request would exceed the installation limits.

If you use VAR=YES when you issue the EXTEND request, use the NUMBLKS parameter to find out the size by which the system extended the data space.

Figure 13-4 on page 13-11 is an example of using the EXTEND request, where the current (and initial) size is 20,000 bytes and the maximum size is 100,000 bytes. If you want to increase the current size to 50,000 bytes, adding 30,000 blocks to the current size, you could code the following:

```
         DSPSERV EXTEND,STOKEN=DSSTOK,BLOCKS=DSPDELTA
          .
DSPDELTA DC  A((30000+4095)/4096)        DATA SPACE ADDITIONAL SIZE
DSSTOK   DS  CL8                        DATA SPACE STOKEN
```

The program can now use 50,000 bytes in the 100,000-byte data space, as shown in Figure 13-5:

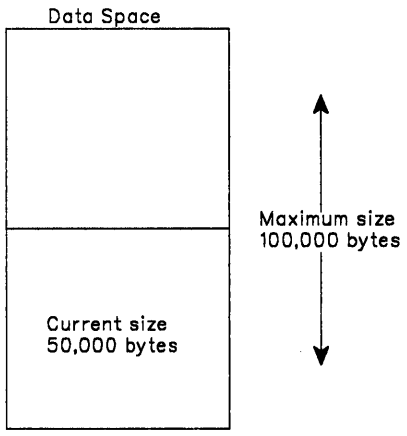


Figure 13-5. Example of Extending the Current Size of a Data Space

Because the example did not include the VAR parameter, the system uses the default, VAR=NO.

Releasing Data Space Storage

Your program needs to release storage when it used a data space for one purpose and wants to reuse it for another purpose, or when your program is finished using the area. To release the virtual storage of a data space, use the DSPSERV RELEASE macro. (Data space release is similar to PGSER RELEASE for an address space.) Specify the STOKEN to identify the data space and the START and BLOCKS parameters to identify the beginning and the length of the area you need to release.

Releasing storage in a data space requires that a problem state program with PSW key 8 - F be the owner or creator of the data space and have the PSW key that matches the storage key of the data space.

Use DSPSERV RELEASE instead of the MVCL instruction to clear large areas of data space storage because:

- DSPSERV RELEASE is faster than MVCL for very large areas.
- Pages released through DSPSERV RELEASE do not occupy space in real, expanded, or auxiliary storage.

Paging Data Space Storage Areas into and out of Central Storage

If you expect to be processing through one or more 4K blocks of data space storage, you can use DSPSERV LOAD to load these pages into central storage. By loading an area of a data space into central storage, you reduce the number of page faults that occur while you sequentially process through that area. DSPSERV LOAD requires that you specify the STOKEN of the data space (on the STOKEN parameter), the beginning address of the area (on the START parameter), and the size of the area (on the BLOCKS parameter). The beginning address has to be on a 4K-byte boundary, and the size has to be an increment of 4K blocks. (Note that DSPSERV LOAD performs the same action for a data spaces as the PGSER macro with the LOAD parameter does for an address space.)

Issuing DSPSERV LOAD does not guarantee that the pages will be in central storage; the system honors your request according to the availability of central storage. Also, after the pages are loaded, page faults might occur elsewhere in the system and cause the system to move those pages out of central storage.

If you finish processing through one or more 4K blocks of data space storage, you can use DSPSERV OUT to page the area out of central storage. The system will make these real storage frames available for reuse. DSPSERV OUT requires that you specify the STOKEN, the beginning address of the area, and the size of the area. (Note that DSPSERV OUT corresponds to the PGSER macro with the OUT parameter.)

When your program has no further need for the data in a certain area of a data space, it can use DSPSERV RELEASE to free that storage.

Deleting a Data Space

When a program doesn't need the data space any more, it should free the virtual storage and remove the entry from the access list. The required parameter on the DSPSERV DELETE macro specifies the STOKEN of the data space to be deleted. A problem state program with PSW key 8 - F must be the owner or creator of the data space and have a PSW key that matches the storage key of the data space.

IBM recommends that you explicitly delete a data space before the owning task terminates. However, if you don't, the system does it for you.

Using Callable Cell Pool Services to Manage Data Space Areas

You can use the callable cell pool services to manage the virtual area in a data space. Callable cell pool services allow you to divide data space storage into areas (cells) of the size you choose. Specifically, you can:

- Create cell pools within a data space
- Expand a cell pool, or make it smaller
- Make the cells available for use by your program or by other programs.

A cell pool consists of one anchor, up to 65,536 extents, and areas of cells, all of which are the same size. The anchor and the extents allow callable cell pool services to keep track of the cell pool.

This section gives an example of one way a program would use the callable cell pool services. This example has only one cell pool with one extent. In the example, you will see that the program has to reserve storage for the anchor and the extent and get their addresses. For more information on how to use the services and an example that includes assembler instructions, see Chapter 10, "Callable Cell Pool Services."

Example of Using Callable Cell Pool Services with a Data Space: Assume that you have an application that requires up to 4,000 records 512 bytes in length. You have decided that a data space is the best place to hold this data. Callable cell pool services can help you build a cell pool, each cell having a size of 512 bytes. The steps are as follows:

1. Create a data space (DSPSERV CREATE macro)

   Specify a size large enough to hold 2,048,000 bytes of data (4000 times 512) plus the data structures that the callable cell pool services need.

2. Add the data space to an access list (ALESERV macro)

   The choice of DU-AL or PASN-AL depends on how you plan to share the data space.

3. Reserve storage for the anchor and obtain its address

   The anchor (of 64 bytes) can be in the address space or the data space. For purposes of this example, the anchor is in the data space.

4. Initialize the anchor (CSRPBLD service) for the cell pool

   Input to CSRPBLD includes the ALET of the data space, the address of the anchor, the name you assign to the pool, and the size of each cell (in this case, 512 bytes). Because the anchor is in the data space, the caller must be in AR mode.

5. Reserve storage for the extent and obtain the address of the extent

   The size of the extent is 128 bytes plus 1 byte for every eight cells. 128 bytes plus 500 (4000 ÷ 8) bytes equals 628 bytes. Callable cell pool services rounds this number to the next doubleword — 632 bytes.

6. Obtain the address of the beginning of the cell storage

   Add the size of the anchor (64 bytes) and the size of the extent (628 bytes) to get the location where the cell storage can start. You might want to make this starting address on a given boundary, such as a doubleword or page.

7. Add an extent for the cell pool and establish a connection between the extent and the cells (CSRPEXP service)

   Input to CSRPEXP includes the ALET for the data space, the address of the anchor, the address of the extent, the size of the extent (in this case, 632 bytes), and the starting address of the cell storage. Because the extent is in the data space, the caller must be in AR mode.

At this point, the cell pool structures are in place and users can begin to request cells. Figure 13-6 describes the areas you have defined in the data space.

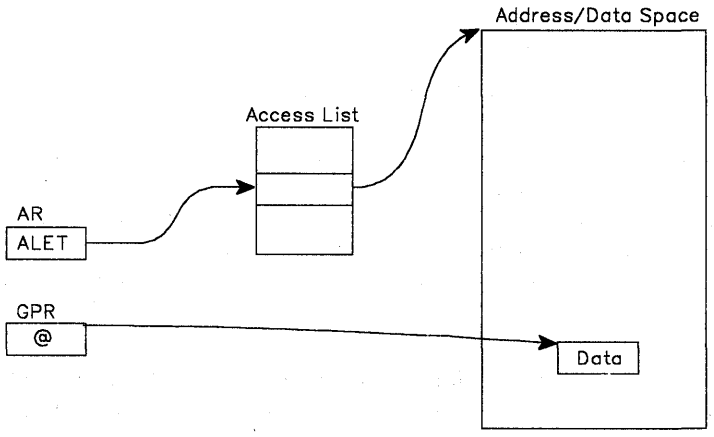


Figure 13-6. Example of Using Callable Cell Pool Services for Data Spaces

A program that has addressability to the data space can then obtain a cell (or cells) through the CSRPGET service. Input to CSRPGET includes the ALET of the space and the address of the anchor. CSRPGET returns the address of the cell (or cells) it allocates.

Programming Notes for the Example:

- The origin of the data space might not be zero for the processor the program is running on. To allow the program to run on more than one processor, use an origin of 4K bytes or use the ORIGIN parameter on DSPSERV to obtain the address of the origin.
- If you need more than one extent, you might have a field that contains the ending address of the last cell pool storage. A program then could use that address to set up another extent and more cells.
- To use callable cell pool services, the caller must be executing in a state or mode or key in which it can write to the storage containing the anchor and the extent data areas.
- The anchor and the extents must be in the same address space or data space. The cells can be in another space.

Sharing Data Spaces among Problem State Programs with PSW Keys 8 - F

Problem state programs with PSW key 8 - F can share data spaces with other programs in several ways:

- A problem state program with PSW key 8 - F can create a data space and place an entry for the data space on its DU-AL. Then the program can attach a subtask and pass a copy of its DU-AL to this subtask, and pass the ALET. However, no existing task in the job step can use this new ALET value.
- A problem state program with PSW key 8 - F can create a data space, add an entry to the PASN-AL, and pass the ALET to other problem state programs running under any task in the job step.
- A problem state program with PSW key 8 - F can create a data space and pass the STOKEN to a program in supervisor state. The supervisor state program can add the entry to either of its access lists.

By attaching a subtask and passing a copy of the DU-AL, a program can share its existing data spaces with a program that runs under the subtask. In this way, the two programs can share the SCOPE=SINGLE data spaces that were represented on the DU-AL at the time of the attach. The ALCOPY=YES parameter on the ATTACH or ATTACHX macro allows a problem state program to pass a **copy** of its DU-AL to the subtask the problem state program is attaching. Passing only a part of the DU-AL is not possible.

For example, as shown in Figure 13-7, assume that program PGM1 (running under TCBA) has created a SCOPE=SINGLE data space DS1 and established addressability to it. PGM1's DU-AL has several entries on it, including one for DS1. PGM1 uses the ATTACHX macro with the ALCOPY=YES parameter to attach subtask TCBB and pass a copy of its DU-AL to TCBB. It can also pass ALETs in a parameter list to PGM2. Upon return from ATTACHX, PGM1 and PGM2 have access to the same data spaces.

The figure shows the two programs, PGM1 and PGM2, sharing the same data space.

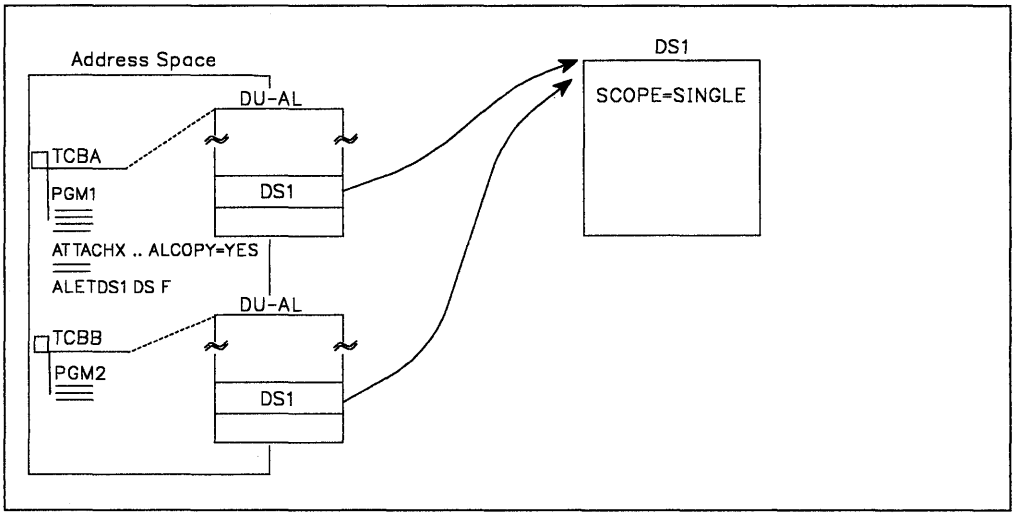


Figure 13-7. Two Problem Programs Sharing a SCOPE=SINGLE Data Space

An example of the code that attaches TCBB and passes a copy of the DU-AL is as follows.

```
   DSPSERV CREATE,NAME=DSNAME,BLOCKS=DSSIZE,STOKEN=DSSTOK,            *
                ORIGIN=DSORG
   ALESERV ADD,STOKEN=DSSTOK,ALET=DSALET
   ATTACHX EP=PGM2,ALCOPY=YES
  .
  .
DSNAME      DC   CL8'TEMP    '   DATA SPACE NAME
DSSTOK      DS   CL8             DATA SPACE STOKEN
DSALET      DS   F               DATA SPACE ALET
DSORG       DS   F               ORIGIN RETURNED
DSSIZE      DC   F'2560'         DATA SPACE 10 MEGABYTES IN SIZE
```

The DU-ALs do not necessarily stay identical. After the attach, PGM1 and PGM2 can add and delete entries on their own DU-ALs; these changes are not made to the other DU-AL.

If TCBA terminates, the system deletes the data space that belonged to TCBA and terminates PGM2.

Sharing Data Spaces through the PASN-AL

One way many problem state programs with PSW key 8 - F can share the data in a data space is by placing the entry for the data space on the PASN-AL and obtaining the ALET. In this way, the programs can pass the ALET to other problem state programs in the address space, allowing them to share the data in the data space.

The following example describes a problem state program with PSW key 8 - F creating a data space and sharing the data in that space with other programs in the address space. Additionally, the program assigns ownership of the data space to its job step task. This assignment allows the data space to be used by other programs even after the creating program's task terminates. In the example, PGM1 creates a 10-megabyte data space named SPACE1. It uses the TTOKEN parameter on DSPSERV to assign ownership to its job step task. Before it issued the DSPSERV CREATE, however, it had to find out the TTOKEN of its job step task. To do this, it issued the TCBTOKEN macro.

```
TCBTOKEN    TTOKEN=JSTTTOK,TYPE=JOBSTEP
 .
DSPSERV CREATE,NAME=DSNAME,BLOCKS=DSSIZE,STOKEN=DSSTOK,ORIGIN=DSORG,
           TTOKEN=JSTTTOK
ALESERV ADD,STOKEN=DSSTOK,ALET=DSALET,AL=PASN
 .
 .
DSNAME      DC  CL8'SPACE1  '   DATA SPACE NAME
DSSTOK      DS  CL8             DATA SPACE STOKEN
DSALET      DS  F               DATA SPACE ALET
DSORG       DS  F               ORIGIN RETURNED
DSSIZE      DC  F'2560'         DATA SPACE 10 MEGABYTES IN SIZE
JSTTTOK     DS  CL8             TTOKEN OF JOB STEP TASK
```

Unless PGM1 or a program running under the job step TCB explicitly deletes the data space, the system deletes the data space when the job step task terminates.

Note that when PGM1 issues the ALESERV ADD to add the entry for DS1 to the PASN-AL, the system checks to see if an entry for DS1 already exists on the PASN-AL. If an entry already exists, and a problem state program with PSW key 8 - F added the entry, the system rejects the ALESERV ADD request. However, PGM1 can still access the data space. The system will simply not create a duplicate entry.

Example of Mapping a Data-in-Virtual Object to a Data Space

Through data-in-virtual, your program can map a VSAM linear data set to a data space. Use DIV macros to set up the relationship between the object and the data space. Setting up this relationship is called "mapping." In this case, the virtual storage area through which you view the object (called the "window") is in the data space. The STOKEN parameter on the DIV MAP macro identifies the data space.

The task that issues the DIV IDENTIFY owns the pointers and structures associated with the ID that DIV returns. Any program can use DIV IDENTIFY; however, the system checks the authority of programs that try to use subsequent DIV services for the same ID.

For problem state programs with PSW key 8 - F, data-in-virtual allows only the issuer of the DIV IDENTIFY to use other DIV services for the ID. That means, for example, that if a problem state program with PSW key 8 issues the DIV IDENTIFY, another problem state program with PSW key 8 cannot issue DIV MAP for the same ID. The issuer of DIV IDENTIFY can use DIV MAP to map a VSAM linear data set to a data space window, providing the program owns or created the data space.

Your program can map one data-in-virtual object into more than one data space. Or, it can map several data-in-virtual objects within a single data space. In this way, data spaces can provide large reference areas available to your program.

Mapping a Data-in-Virtual Object to a Data Space

The following example maps a data-in-virtual object in a data space. The size of the data space is 10 megabytes, or 2560 blocks. (A block is 4K bytes.)

```
* CREATE A DATA SPACE, ADD AN ACCESS LIST ENTRY FOR IT
* AND MAP A DATA-IN-VIRTUAL OBJECT INTO DATA SPACE STORAGE
  .
  DSPSERV CREATE,NAME=DSNAME,STOKEN=DSSTOK,BLOCKS=DSSIZE,ORIGIN=DSORG
  ALESERV ADD,STOKEN=DSSTOK,ALET=DSALET,AL=WORKUNIT,ACCESS=PUBLIC
  .
* EQUATE DATA SPACE STORAGE TO OBJAREA
  .
  L       4,DSORG
  LAM     4,4,DSALET
  USING   OBJAREA,4
  .
* MAP THE OBJECT
  .
  DIV     IDENTIFY,ID=OBJID,TYPE=DA,DDNAME=OBJDD
  DIV     ACCESS,ID=OBJID,MODE=UPDATE
  DIV     MAP,ID=OBJID,AREA=OBJAREA,STOKEN=DSSTOK
  .
* USE THE ALET IN DSALET TO REFERENCE THE
* DATA SPACE STORAGE MAPPING THE OBJECT.

  .
* SAVE ANY CHANGES TO THE OBJECT WITH DIV SAVE
  .
  DIV     SAVE,ID=OBJID
  DIV     UNMAP,ID=OBJID,AREA=DSORG
  DIV     UNACCESS,ID=OBJID
  DIV     UNIDENTIFY,ID=OBJID
  .
* DELETE THE ACCESS LIST ENTRY AND THE DATA SPACE
  .
  ALESERV DELETE,ALET=DSALET
  DSPSERV DELETE,STOKEN=DSSTOK
  .
DSNAME   DC   CL8'MYSPACE '        DATA SPACE NAME
DSSTOK   DS   CL8                  DATA SPACE STOKEN
DSALET   DS   F                    DATA SPACE ALET
DSORG    DS   F                    DATA SPACE ORIGIN
DSSIZE   DC   F'2560'              DATA SPACE 10 MEGABYTES IN SIZE
OBJID    DS   CL8                  DIV OBJECT ID
OBJDD    DC   AL1(7),CL7'MYDD   '  DIV OBJECT DDNAME
OBJAREA  DSECT                     WINDOW IN DATA SPACE
OBJWORD1 DS   F
OBJWORD2 DS   F
```

Using Data Spaces Efficiently

Although a task can own many data spaces, it is important that it reference these data spaces carefully. It is much more efficient for the system to reference the same data space ten times than it is to reference each of ten data spaces one time. For example, an application might have a master application region that has many users, each one having a data space. If each program completes its work with one data space before it starts work with another data space, performance is optimized.

Example of Creating, Using, and Deleting a Data Space

This section contains an example of a problem state program creating, establishing addressability to, using, and deleting the data space named TEMP. The first lines of code create the data space and establish addressability to the data space. To keep the example simple, the code does not include the checking of the return code from the DSPSERV macro or the ALESERV macro. You should, however, always check return codes.

The lines of code in the middle of the example illustrate how, with the code in AR mode, the familiar assembler instructions store, load, and move a simple character string into the data space and move it within the data space. The example ends with the program deleting the data space entry from the access list, deleting the data space, and returning control to the caller.

```
DSPEXMPL CSECT
DSPEXMPL AMODE 31
DSPEXMPL RMODE ANY
         BAKR  14,0                    SAVE CALLER'S STATUS ON STACK
         SAC   512                     SWITCH INTO AR MODE
         SYSSTATE ASCENV=AR            ENSURE PROPER CODE GENERATION
         .
         LAE   12,0                    SET BASE REGISTER AR
         BASR  12,0                    SET BASE REGISTER GPR
         USING *,12

         .
         DSPSERV CREATE,NAME=DSPCNAME,STOKEN=DSPCSTKN,           X
               BLOCKS=DSPBLCKS,ORIGIN=DSPCORG
         ALESERV ADD,STOKEN=DSPCSTKN,ALET=DSPCALET,AL=WORKUNIT
         .
*  ESTABLISH ADDRESSABILITY TO THE DATA SPACE
         .
         LAM   2,2,DSPCALET            LOAD ALET OF SPACE INTO AR2
         L     2,DSPCORG               LOAD ORIGIN OF SPACE INTO GPR2
         USING DSPCMAP,2               INFORM ASSEMBLER
         .
*  MANIPULATE DATA IN THE DATA SPACE
         .
         L     3,DATAIN
         ST    3,DSPWRD1               STORE INTO DATA SPACE WRD1
         .
         MVC   DSPWRD2,DATAIN          COPY DATA FROM PRIMARY SPACE
*                                       INTO THE DATA SPACE
         MVC   DSPWRD3,DSPWRD2         COPY DATA FROM ONE LOCATION
*                                       IN THE DATA SPACE TO ANOTHER
         MVC   DATAOUT,DSPWRD3         COPY DATA FROM DATA SPACE
*                                       INTO THE PRIMARY SPACE
         .
*  DELETE THE ACCESS LIST ENTRY AND THE DATA SPACE
         .
         ALESERV DELETE,ALET=DSPCALET    REMOVE DS FROM AL
         DSPSERV DELETE,STOKEN=DSPCSTKN  DELETE THE DS
         .
         PR                              RETURN TO CALLER
         .
```

```
         DS   0D
DSPCNAME DC   CL8'TEMP     '            DATA SPACE NAME
DSPCSTKN DS   CL8                       DATA SPACE STOKEN
DSPCALET DS   F                         DATA SPACE ALET
DSPCORG  DS   F                         DATA SPACE ORIGIN RETURNED
DSPCSIZE EQU  10000000                  10 MILLION BYTES OF SPACE
DSPBLCKS DC   A((DSPCSIZE+4095)/4096)   NUMBER OF BLOCKS NEEDED FOR
*                                       A 10 MILLION BYTE DATA SPACE
DATAIN   DC   CL4'ABCD'
DATAOUT  DS   CL4
         .
DSPCMAP  DSECT              MAPPING OF DATA SPACE STORAGE
DSPWRD1  DS    F            WORD 1
DSPWRD2  DS    F            WORD 2
DSPWRD3  DS    F            WORD 3
         END
```

Note that you cannot code ACCESS=PRIVATE on the ALESERV macro when you request an ALET for a data space; all data space entries are public.

Dumping Storage in a Data Space

MVS provides ways to dump areas of data space storage. You can use the SNAPX macro. You can also ask the operator to use the SLIP and DUMP commands. After the system enters a wait state or hangs or enters a loop, the operator can request a stand-alone dump.

On the SNAPX macro,

- Use the DSPSTOR parameter on the SNAPX macro to dump storage from any addressable data space that the caller can access.

For the syntax of SNAPX, see *Assembler Programming Reference*.

On the SLIP command,

- The DSPNAME parameter includes data space storage in a dump.

On the DUMP command,

- An operator can use the DSPNAME parameter on the DUMP command to dump all of the storage in a number of data spaces.

For the syntax of the SLIP and DUMP commands, see *System Commands*.

To request a stand-alone dump:

- An operator can request that the stand-alone program, AMDSADMP, dump the storage in a data space. Use the DATASPACES keyword on the SADMP DUMP command.

For more information about the DATASPACES keyword, see *Service Aids*.

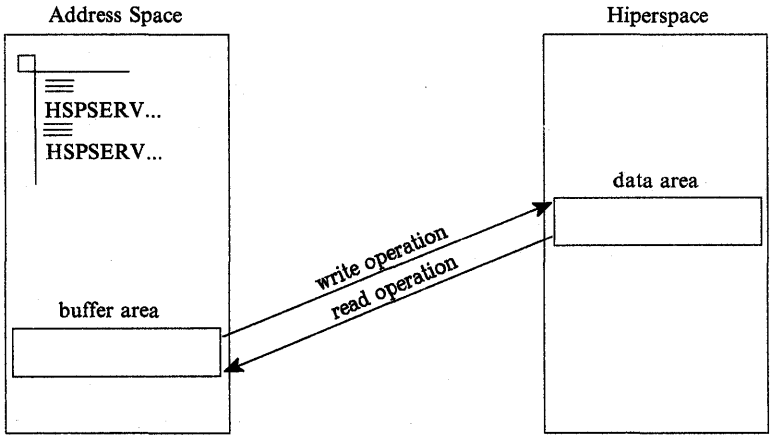
Creating and Using Hiperspaces

A **hiperspace** is a range of up to two gigabytes of contiguous virtual storage addresses that a program can use as a buffer. Like a data space, a hiperspace holds only data, not common areas or system data; code does not execute in a hiperspace. Unlike data in a data space, data in a hiperspace is not directly addressable.

The DSPSERV macro manages hiperspaces. The TYPE=HIPERSPACE parameter tells the system that it is to manage a hiperspace rather than a data space. Use DSPSERV to:

- Create a hiperspace
- Release an area in a hiperspace
- Delete a hiperspace
- Expand the amount of storage in a hiperspace currently available to a program.

To manipulate data in a hiperspace, your program brings the data, in blocks of 4K bytes, into a buffer area in the address space. The program can use the data only while it is in the address space. You can think of this buffer area as a "view" into the hiperspace. The HSPSERV macro read operation manages the transfer of the data to the address space buffer area. If you make updates to the data, you can write it back to the hiperspace through the HSPSERV write operation.



The data in the hiperspace and the buffer area in the address space must both start on a 4K byte boundary.

Use this section to help you create, use, and delete hiperspaces. It describes some of the characteristics of hiperspaces, how to move data in and out of a hiperspace; and how data-in-virtual can help you control data in hiperspaces. In addition, *Assembler Programming Reference* contains the syntax and parameter descriptions for the macros that are mentioned in this section.

Standard Hiperspaces

Your program can create a standard hiperspace, one that is backed with expanded storage and auxiliary storage, if necessary. Through the buffer area in the address space, your program can view or "scroll" through the hiperspace. Scrolling allows you to make interim changes to data without changing the data on DASD. HSTYPE=SCROLL on DSPSERV creates a standard hiperspace. HSPSERV SWRITE and HSPSERV SREAD transfer data to and from a standard hiperspace.

The data in a standard hiperspace is predictable; that is, your program can write data to a standard hiperspace and count on retrieving it.

The best way to describe how your program can scroll through a standard hiperspace is through an example. Figure 13-8 shows a hiperspace that has four scroll areas, A, B, C, and D. After the program issues an HSPSERV SREAD for hiperspace area A, it can make changes to the data in the buffer area in its address space. HSPSERV SWRITE then saves those changes. In a similar manner, the program can read, make changes, and save the data in areas B, C, and D. When the program reads area A again, it finds the same data that it wrote to the area in the previous HSPSERV SWRITE to that area.

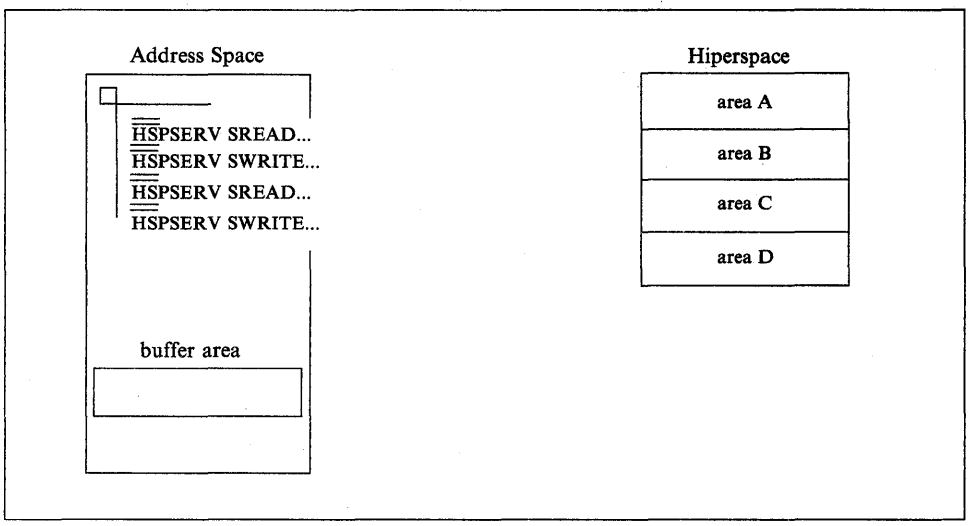


Figure 13-8. Example of Scrolling through a Standard Hiperspace

A standard hiperspace gives your program an area where it can:

- Store data, either generated by your program or moved (through the address space buffers) from DASD
- Scroll through large amounts of data

After you finish using the hiperspace, you can:

- Move the changed data (through address space buffers) to DASD, making the hiperspace data permanent
- Delete the hiperspace data with the deletion of the hiperspace or the termination of the owner of the hiperspace, treating the hiperspace data as temporary.

If your application wants to save a permanent copy of the data in the hiperspace, consider using the services of data-in-virtual. See "Using Data-in-Virtual with Hiperspaces" on page 13-37.

A second type of hiperspace, the expanded storage only (ESO) hiperspace is backed with expanded storage only and is available to supervisor state programs or programs with PSW key 0 - 7. These hiperspaces are described in the books that are available to writers of authorized programs.

Shared and Non-shared Standard Hiperspaces

Standard hiperspaces are either non-shared or shared. Your program can create and delete **non-shared standard hiperspaces**; it can use HSPSERV to access the non-shared standard hiperspaces that it owns. With help from a supervisor state program or a program with PSW key 0 - 7, your program can also access a non-shared standard hiperspaces that it does not own. **Shared standard hiperspaces** can be shared among programs in many address spaces. Although your programs can use the shared standard hiperspaces, they cannot create and delete them. Therefore, the sharing of hiperspaces must be done under the control of supervisor state programs or programs with PSW key 0 - 7. Shared standard hiperspaces and the subject of sharing hiperspaces are described in the application development books that are available to writers of authorized programs. The chapter in this book describes how you create and delete the non-shared standard hiperspaces and use these hiperspaces for your own program.

Figure 13-9 shows some important facts about non-shared standard hiperspaces:

Figure 13-9. Facts about a Non-shared Standard Hiperspace

Can it map a VSAM linear data set?	Yes
Can it be a data-in-virtual object?	Yes, if the hiperspace has not been the target of ALESERV ADD.
How do you write data to the hiperspace?	By using HSPSERV SWRITE
How do you read data from the hiperspace?	By using HSPSERV SREAD
What happens to the data in the hiperspace when the system swaps the owning address space out?	The system preserves the data.

Creating a Hiperspace

To create a non-shared standard hiperspace, issue the DSPSERV CREATE macro with the TYPE=HIPERSPACE and HSTYPE=SCROLL parameters. The HSTYPE parameter tells the system you want a standard hiperspace. HSTYPE=SCROLL is the default. MVS allocates contiguous 31-bit virtual storage of the size you specify and initializes the storage to hexadecimal zeroes. The entire hiperspace has the storage key 8. Because many of the same rules that apply to creating data spaces also apply to creating hiperspaces, this section sometimes refers you to sections earlier in "Creating a Data Space."

On the DSPSERV macro, you are required to specify:

- The name of the hiperspace (NAME parameter)

  To ask DSPSERV to generate a hiperspace name unique to the address space, use the GENNAME parameter. DSPSERV will return the name it generates at the location you specify on the OUTNAME parameter. Specifying a name for a hiperspace follows the same rules as specifying a name for a data space. See "Choosing the Name of a Data Space" on page 13-9.

- A location where DSPSERV is to return the STOKEN of the hiperspace (STOKEN parameter)

  DSPSERV CREATE returns a STOKEN that you can use to identify the hiperspace to other DSPSERV services and to the HSPSERV and DIV macros.

Other information you might specify on the DSPSERV macro is:

- The maximum size of the hiperspace and its initial size (BLOCKS parameter). If you do not code BLOCKS, the hiperspace size is determined by defaults set by your installation. In this case, use the NUMBLKS parameter to tell the system where to return the size of the hiperspace. Specifying the size of a hiperspace follows the same rules as specifying the size of a data space. See "Specifying the Size of a Data Space" on page 13-10.
- A location where DSPSERV can return the address (either 0 or 4096) of the first available block of the hiperspace (ORIGIN parameter). Locating the origin of a hiperspace is the same as locating the origin of a data space. See "Identifying the Origin of a Data Space" on page 13-11.

Example of Creating a Standard Hiperspace

The following example creates a non-shared standard hiperspace, 20 blocks in size, named SCROLLHS.

```
*
         DSPSERV CREATE,NAME=HSNAME,TYPE=HIPERSPACE,HSTYPE=SCROLL,     X
               BLOCKS=20,STOKEN=HSSTOKEN
*
HSNAME   DC   CL8'SCROLLHS'          * NAME FOR THE HIPERSPACE
HSSTOKEN DS   CL8                    * STOKEN OF THE HIPERSPACE
```

Transferring Data To and From Hiperspaces

Before it can reference data or manipulate data in a hiperspace, the program must bring the data into the address space. The HSPSERV macro performs the transfer of data between the address space and the hiperspace.

On the HSPSERV macro, the **write operation** transfers data from the address space to the hiperspace. The **read operation** transfers the data from the hiperspace to the address space. HSPSERV allows multiple reads and writes to occur at one time. This means that one HSPSERV request can read more than one data area in a hiperspace to an equal number of data areas in an address space. Likewise, one HSPSERV request can write data from more than one buffer area in an address space to an equal number of areas in a hiperspace.

Figure 13-10 shows three virtual storage areas that you identify on the HSPSERV macro when you request a data transfer:

- The hiperspace
- The buffer area in the address space that is the source of the write operation and the target of the read operation
- The data area in the hiperspace that is the target of the write operation and the source of the read operation

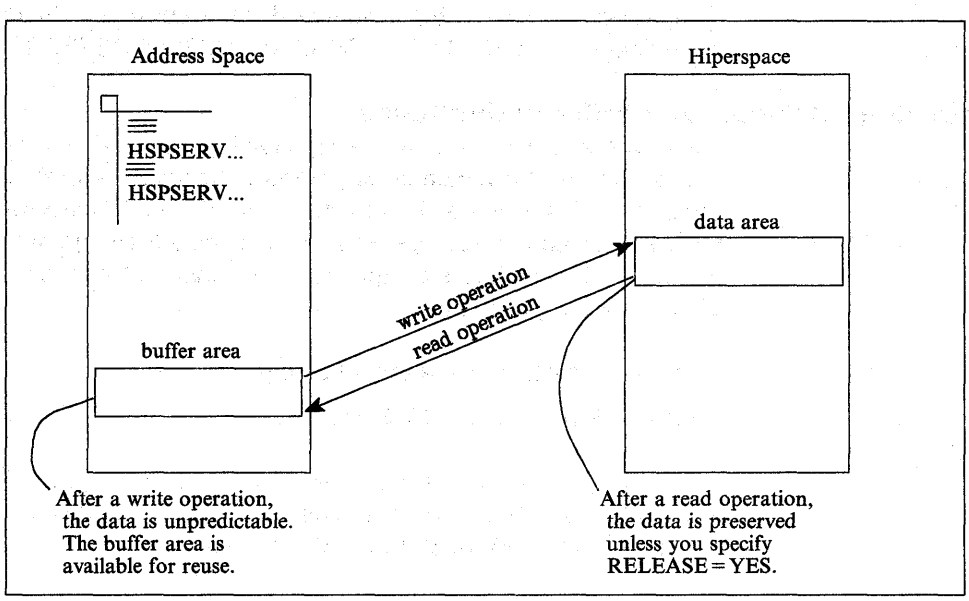


Figure 13-10. Illustration of the HSPSERV Write and Read Operations

On the HSPSERV macro, you identify the hiperspace and the areas in the address space and the hiperspace:

- STOKEN specifies the STOKEN of the hiperspace.
- NUMRANGE specifies the number of data areas the system is to read or write.
- RANGLIST specifies a list of ranges that indicate the boundaries of the buffer areas in the address space and the data area in the hiperspace.

HSPSERV has certain restrictions on these areas. Two restrictions are that the data areas must start on a 4K byte boundary and their size must be in multiples of 4K bytes. Other requirements are listed in the description of HSPSERV in *Assembler Programming Reference*. Read the requirements carefully before you issue the macro.

The system does not always preserve the data in the areas that are the source for the read and write operations. Figure 13-10 tells you what the system does with the areas after it completes the transfer.

Read and Write Operations for Standard Hiperspaces

After the write operation for standard hiperspaces, the system does not preserve the data in the address space. It assumes that you have another use for that buffer area, such as using it as the target of another HSPSERV SREAD operation.

After the read operation for standard hiperspaces, the system gives you a choice of saving the source data in the hiperspace. If you will use the data in the hiperspace again, ask the system to preserve the data; specify RELEASE=NO on HSPSERV SREAD. Unless a subsequent SWRITE request changes the data in the source area, that same data will be available for subsequent SREAD requests. RELEASE=NO provides your program with a backup copy of the data in the hiperspace.

If you specify RELEASE=YES on HSPSERV SREAD, the system releases the hiperspace pages after the read operation and returns the expanded storage (or auxiliary storage) that backs the source area in the hiperspace. RELEASE=YES tells the system that your program does not plan to use the source area in the hiperspace as a copy of the data after the read operation.

See "Example of Creating a Standard Hiperspace and Using It" on page 13-35 for an example of the HSPSERV SREAD and HSPSERV SWRITE macros.

Obtaining Additional HSPSERV Performance

If your processor has the move-page facility installed, you can use HSPSERV to improve the performance of data transfer between central and expanded storage. Specify the ALET of the hiperspace on the HSPALET parameter on HSPSERV. If the data is in expanded storage, HSPSERV takes advantage of the move-page facility. If the data is in auxiliary storage, the data transfer still occurs, but without the benefit of the move-page facility.

To obtain the ALET, issue the following:

```
ALESERV ADD,ALET=. . .,STOKEN=. . .
```

STOKEN is the eight-byte identifier of the hiperspace, and ALET is the four-byte index into the DU-AL, the access list that is associated with the task. The STOKEN is input to ALESERV ADD; the ALET is output.

Before you issue the HSPSERV macro with the HSPALET parameter, obtain a 144-byte workarea for the HSPSERV macro service and place the address of this area in GPR 13 and a zero in AR 13

Do not specify RELEASE=YES with the HSPALET parameter.

Programming Notes for Obtaining ALETs for Hiperspaces

- To use ALESERV ADD to obtain an ALET for a hiperspace without having the move-page facility installed causes the program to abend.
- A program never uses an ALET to directly access data in a hiperspace as it would use the ALET to access the data in a data space.
- To use hiperspaces, you do not need to switch into AR mode.
- When you are finished using the hiperspace, use ALESERV DELETE to delete the entry on the DU-AL.
- The system places certain restrictions on the combined use of hiperspaces and data-in-virtual. These restrictions are listed in "Using Data-in-Virtual with Hiperspaces" on page 13-37.
- By obtaining an ALET, you can share a hiperspace with a subtask in the same way you share a data space. Use the ALCOPY parameter on the ATTACHX macro to pass a copy of your DU-AL to the subtask. Follow the procedure suggested in "Sharing Data Spaces among Problem State Programs with PSW Keys 8 - F" on page 13-20.

Example of a HSPSERV with the HSPALET Parameter: The following example creates a non-shared hiperspace. To get additional performance from HSPSERV, the program obtains an ALET from the ALESERV macro and uses that ALET as input to HSPSERV. The example assumes the ASC mode is primary.

```
:
* DSPSERV CREATES A NON-SHARED STANDARD HIPERSPACE OF 20 4096-BYTE BLOCKS
*
        DSPSERV CREATE,NAME=HSNAME,TYPE=HIPERSPACE,BLOCKS=20,         X
              STOKEN=HSSTOKEN,ORIGIN=HSORIG1
*
* ALESERV RETURNS AN ALET ON THE DU-AL FOR THE HIPERSPACE
*
        ALESERV ADD,STOKEN=HSSTOKEN,ALET=HSALET,AL=WORKUNIT
*
* THE STORAGE MACRO OBTAINS FOUR PAGES OF ADDRESS SPACE STORAGE,
* THE BNDRY=PAGE PARAMETER ALIGNS PAGES ON A 4K BOUNDARY
*   - THE FIRST AND SECOND PAGES ARE THE SWRITE SOURCE
*   - THE THIRD AND FOURTH PAGES ARE THE SREAD TARGET
* COPY INTO FIRST AND SECOND PAGES THE DATA TO BE WRITTEN TO HIPERSPACE
   .
        STORAGE OBTAIN,LENGTH=4096*4,BNDRY=PAGE
        ST   1,ASPTR              * SAVE ADDR SPACE STORAGE ADDRESS
        MVC  0(20,1),SRCTEXT1     * INIT FIRST ADDR SPACE PAGE
        A    1,ONEBLK             * COMPUTE PAGE TWO ADDRESS
        MVC  0(20,1),SRCTEXT2     * INIT SECOND ADDR SPACE PAGE
   .
* SET UP THE SWRITE RANGE LIST TO WRITE FROM THE FIRST AND SECOND
* ADDRESS SPACE PAGES INTO THE HIPERSPACE
   .
        L    1,ASPTR              * GET FIRST ADDR PAGE ADDRESS
        ST   1,ASPTR1             * PUT ADDRESS INTO RANGE LIST
   .
* SAVE CONTENTS OF AR/GPR 13 BEFORE RESETTING THEM FOR HSPSERV
   .
        ST   13,SAVER13           * SAVE THE CONTENTS OF GPR 13
        EAR  13,13                * LOAD GPR 13 FROM AR 13
        ST   13,SAVEAR13          * SAVE THE CONTENTS OF AR 13
   .
* ESTABLISH ADDRESS OF 144-BYTE SAVE AREA, AS HSPALET ON HSPSERV REQUIRES
* AND WRITE TWO PAGES FROM THE ADDRESS SPACE TO THE HIPERSPACE
   .
        SLR  13,13                * SET GPR 13 TO 0
        SAR  13,13                * SET AR 13 TO 0
        LA   13,WORKAREA          * SET UP AR/GPR 13 TO WORKAREA ADDR
        HSPSERV SWRITE,STOKEN=HSSTOKEN,RANGLIST=RANGPTR1,HSPALET=HSALET
   .
* AFTER THE SWRITE, THE FIRST TWO ADDRESS SPACE PAGES MIGHT BE OVERLAID
   .
* RESTORE ORIGINAL CONTENTS OF AR/GPR 13
   .
        L    13,SAVEAR13          * SET GPR 13 TO SAVED AR 13
        SAR  13,13                * RESET AR 13
        L    13,SAVER13           * RESET GPR 13
   .
* SET UP THE SREAD RANGE LIST TO READ INTO THE THIRD AND FOURTH
* ADDRESS SPACE PAGES WHAT WAS PREVIOUSLY WRITTEN TO THE HIPERSPACE
   .
        MVC  HSORIG2,HSORIG1     * COPY ORIGIN OF HIPERSPACE TO HSORIG2
        L    1,ASPTR              * GET FIRST ADDR PAGE ADDRESS
        A    1,TWOBLKS            * COMPUTE THIRD PAGE ADDRESS
        ST   1,ASPTR2             * PUT ADDRESS INTO RANGE LIST
```

```
*
* SAVE CONTENTS OF AR/GPR 13
*
         ST    13,SAVER13           * SAVE THE CONTENTS OF GPR 13
         EAR   13,13                * LOAD GPR 13 FROM AR 13
         ST    13,SAVEAR13          * SAVE THE CONTENTS OF AR 13
*
* ESTABLISH ADDRESS OF 144-BYTE SAVE AREA, AS HSPALET ON HSPSERV REQUIRES,
* AND READ TWO BLOCKS OF DATA FROM THE HIPERSPACE INTO THE
* THIRD AND FOURTH PAGES IN THE ADDRESS SPACE STORAGE USING HSPALET
*
         SLR   13,13                * SET GPR 13 TO 0
         SAR   13,13                * SET AR 13 TO 0
         LA    13,WORKAREA          * SET UP AR/GPR 13 TO WORKAREA ADDR
         HSPSERV SREAD,STOKEN=HSSTOKEN,RANGLIST=RANGPTR2,HSPALET=HSALET
*
* RESTORE ORIGINAL CONTENTS OF AR/GPR 13
*
         L     13,SAVEAR13          * SET GPR 13 TO SAVED AR 13
         SAR   13,13                * RESET AR 13
         L     13,SAVER13           * RESET GPR 13
*
* FREE THE ALET, FREE ADDRESS SPACE STORAGE, AND DELETE THE HIPERSPACE
⋮
* DATA AREAS AND CONSTANTS
*
HSNAME   DC    CL8'SCROLLHS'        * NAME FOR THE HIPERSPACE
HSSTOKEN DS    CL8                  * STOKEN FOR THE HIPERSPACE
HSALET   DS    CL4                  * ALET FOR THE HIPERSPACE
ASPTR    DS    1F                   * LOCATION OF ADDR SPACE STORAGE
SAVER13  DS    1F                   * LOCATION TO SAVE GPR 13
SAVEAR13 DS    1F                   * LOCATION TO SAVE AR 13
WORKAREA DS    CL144                * WORK AREA FOR HSPSERV
ONEBLK   DC    F'4096'              * LENGTH OF ONE BLOCK OF STORAGE
TWOBLKS  DC    F'8092'              * LENGTH OF TWO BLOCKS OF STORAGE
SRCTEXT1 DC    CL20' INVENTORY ITEMS    '
SRCTEXT2 DC    CL20' INVENTORY SURPLUSES'
         DS    0F
RANGPTR1 DC    A(SWRITLST)          * ADDRESS OF SWRITE RANGE LIST
RANGPTR2 DC    A(SREADLST)          * ADDRESS OF SREAD RANGE LIST
         DS    0F
SWRITLST DS    0CL12                * SWRITE RANGE LIST
ASPTR1   DS    F                    * START OF ADDRESS SPACE SOURCE
HSORIG1  DS    F                    * TARGET LOCATION IN HIPERSPACE
NUMBLKS1 DC    F'2'                 * NUMBER OF 4K BLOCKS IN SWRITE
         DS    0F
SREADLST DS    0CL12                * SREAD RANGE LIST
ASPTR2   DS    F                    * TARGET LOCATION IN ADDR SPACE
HSORIG2  DS    F                    * START OF HIPERSPACE SOURCE
NUMBLKS2 DC    F'2'                 * NUMBER OF 4K BLOCKS IN SREAD
         DS    0F
```

Extending the Current Size of a Hiperspace

When you create a hiperspace and specify a maximum size larger than the initial size, you can use DSPSERV EXTEND to increase the current size as your program uses more storage in the hiperspace. The BLOCKS parameter specifies the amount of storage you want to add to the current size of the hiperspace. The VAR parameter tells the system whether the request is variable. For information about a variable request and help in using DSPSERV EXTEND, see "Extending the Current Size of a Data Space" on page 13-16.

Releasing Hiperspace Storage

Your program needs to release storage when it used a hiperspace for one purpose and wants to reuse it for another purpose, or when your program is finished using the area. To release the virtual storage of a hiperspace, use the DSPSERV RELEASE macro. (Hiperspace release is similar to a PGSER RELEASE for an address space.) Specify the STOKEN to identify the hiperspace and the START and BLOCKS parameters to identify the beginning and the length of the area you need to release.

Releasing storage in a hiperspace requires that a program have the following authority:

- The program must be the owner of the hiperspace.
- The program's PSW key must equal the storage key of the hiperspace the system is to release. Otherwise, the system abends the caller.

After the release, a released page does not occupy expanded (or auxiliary) storage until your program references it again. When you again reference a page you have released, the page contains hexadecimal zeroes.

Use DSPSERV RELEASE instead of the MVCL instruction to clear 4K byte blocks of storage to zeroes because:

- DSPSERV RELEASE is faster than MVCL for very large areas.
- Pages released through DSPSERV RELEASE do not occupy space in expanded or auxiliary storage.

Deleting a Hiperspace

When a program doesn't need the hiperspace any more, it can delete it. Your program can delete only the hiperspaces it owns, providing the program's PSW key matches the storage key of the hiperspace.

Example of Deleting a Hiperspace: The following example shows you how to delete a hiperspace:

```
         DSPSERV DELETE,STOKEN=HSSTKN       DELETE THE HS
         .
HSSTKN   DS   CL8                           HIPERSPACE STOKEN
```

IBM recommends that you explicitly delete a hiperspace before the owning task terminates. However, if you don't, the system automatically does it for you.

Example of Creating a Standard Hiperspace and Using It

The following example creates a standard hiperspace named SCROLLHS. The size of the hiperspace is 20 blocks. The program:

- Creates a standard hiperspace 20 blocks in size
- Obtains four pages of address space storage aligned on a 4K byte address
- Sets up the SWRITE range list parameter area to identify the first two pages of the address space storage
- Initializes the first two pages of address space storage that will be written to the hiperspace
- Issues the HSPSERV SWRITE macro to write the first two pages to locations 4096 through 12287 in the hiperspace

Later on, the program:

- Sets up the SREAD range list parameter area to identify the last two pages of the four-page address space storage
- Issues the HSPSERV SREAD macro to read the blocks at locations 4096 through 12287 in the hiperspace to the last two pages in the address space storage

Figure 13-11 shows the four-page area in the address space and the two block area in the hiperspace. Note that the first two pages of the address space virtual storage are unpredictable after the SWRITE operation.

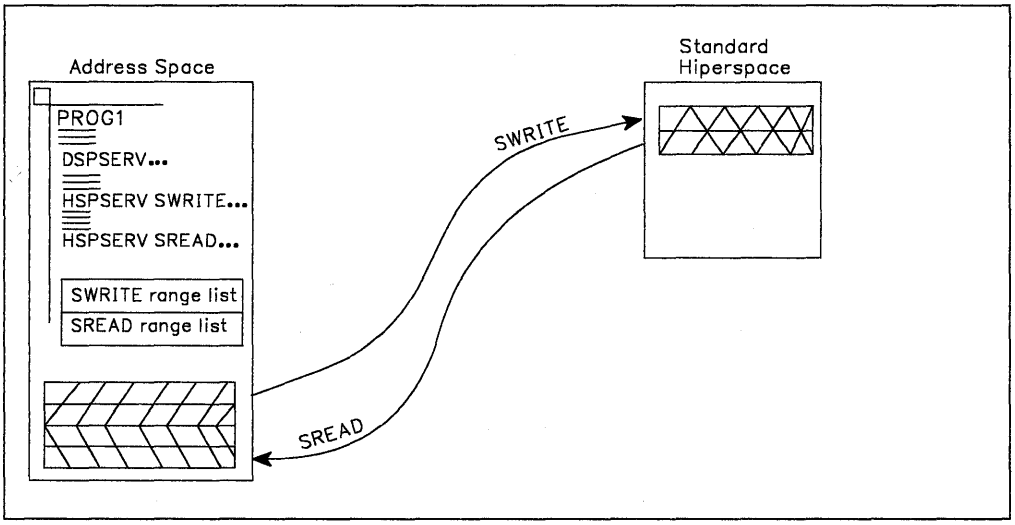


Figure 13-11. Example of Creating a Standard Hiperspace and Transferring Data

```
* DSPSERV CREATES A STANDARD TYPE HIPERSPACE OF 20 4096-BYTE BLOCKS
         .
         DSPSERV CREATE,NAME=HSNAME,TYPE=HIPERSPACE,HSTYPE=SCROLL,    X
               BLOCKS=20,STOKEN=HSSTOKEN
         .
* THE STORAGE MACRO OBTAINS FOUR PAGES OF ADDRESS SPACE STORAGE.
* THE BNDRY=PAGE PARAMETER ALIGNS PAGES ON A 4K BOUNDARY
*  - THE FIRST AND SECOND PAGES ARE THE SWRITE SOURCE
*  - THE THIRD AND FOURTH PAGES ARE THE SREAD TARGET
         .
         STORAGE OBTAIN,LENGTH=4096*4,BNDRY=PAGE
         ST   1,ASPTR1              * SAVES THE SWRITE SOURCE ADDRESS
         MVC  0(20,1),SRCTEXT1      * INITIALIZES SOURCE PAGE ONE
         A    1,ONEBLOCK            * COMPUTES SOURCE PAGE TWO ADDRESS
         MVC  0(20,1),SRCTEXT2      * INITIALIZES SOURCE PAGE TWO
         .
* HSPSERV WRITES TWO PAGES FROM THE ADDRESS SPACE TO THE HIPERSPACE
         .
         HSPSERV SWRITE,STOKEN=HSSTOKEN,RANGLIST=RANGPTR1
         .
* AFTER THE SWRITE, THE FIRST TWO ADDRESS SPACE PAGES MIGHT BE OVERLAID
         .
         .
* SET UP THE SREAD RANGE LIST TO READ INTO THE THIRD AND FOURTH
* ADDRESS SPACE PAGES
         .
         L    2,ASPTR1              * OBTAINS THE ADDRESS OF PAGE 1
         A    2,ONEBLOCK            * COMPUTES THE SREAD TARGET ADDRESS
         A    2,ONEBLOCK            * COMPUTES THE SREAD TARGET ADDRESS
         ST   2,ASPTR2              * SAVES IN SREAD RANGE LIST
         .
* HSPSERV READS TWO BLOCKS OF DATA FROM THE HIPERSPACE TO THE
   THIRD AND FOURTH PAGES IN THE ADDRESS SPACE STORAGE
         .
         HSPSERV SREAD,STOKEN=HSSTOKEN,RANGLIST=RANGPTR2
         .
* DATA AREAS AND CONSTANTS
*
HSNAME   DC   CL8'SCROLLHS'         * NAME FOR THE HIPERSPACE
HSSTOKEN DS   CL8                   * STOKEN FOR THE HIPERSPACE
ONEBLOCK DC   F'4096'                * LENGTH OF ONE BLOCK OF STORAGE
SRCTEXT1 DC   CL20' INVENTORY ITEMS    '
SRCTEXT2 DC   CL20' INVENTORY SURPLUSES'
         DS   0F
RANGPTR1 DC   A(SWRITLST)           * ADDRESS OF THE SWRITE RANGE LIST
RANGPTR2 DC   A(SREADLST)           * ADDRESS OF THE SREAD RANGE LIST

         DS   0F
SWRITLST DS   0CL12                 *  SWRITE RANGE LIST
ASPTR1   DS   F                     *  START OF ADDRESS SPACE SOURCE
HSPTR1   DC   F'4096'                *  TARGET LOCATION IN HIPERSPACE
NUMBLKS1 DC   F'2'                  *  NUMBER OF 4K BLOCKS IN SWRITE
         DS   0F
SREADLST DS   0CL12                 *  SREAD RANGE LIST
ASPTR2   DS   F                     *  TARGET LOCATION IN ADDRESS SPACE
HSPTR2   DC   F'4096'                *  START OF HIPERSPACE SOURCE
NUMBLKS2 DC   F'2'                  *  NUMBER OF 4K PAGES IN SREAD
```

Using Data-in-Virtual with Hiperspaces

Data-in-virtual allows you to map a large amount of data into a virtual storage area and then deal with the portion of the data that you need. The virtual storage provides a "window" through which you can "view" the object and make changes, if you want. The DIV macro manages the data object, the window, and the movement of data between the window and the object.

You can use standard hiperspaces with data-in-virtual in two ways:

1. You can map a VSAM linear data set to hiperspace virtual storage.
2. You can map a non-shared hiperspace to address space virtual storage.

The task that issues the DIV IDENTIFY owns the pointers and structures associated with the ID that DIV returns. Any program can use DIV IDENTIFY. However, the system checks the authority of programs that try to use the other DIV services for the same ID. For problem state programs with PSW key 8 - F, data-in-virtual allows only the issuer of the DIV IDENTIFY to use subsequent DIV services for the same ID. That means, for example, that if a problem state program with PSW key 8 issues the DIV IDENTIFY, another problem state program with PSW key 8 cannot issue DIV MAP for the same ID.

Problem state programs with PSW key 8 - F can use DIV MAP to:

- Map a VSAM linear data set to a window in a hiperspace, providing the program owns the hiperspace.
- Map a non-shared hiperspace object to an address space window, providing:
  - The program owns the hiperspace,
  - The program or its attaching task obtained the storage for the window, and
  - No program has ever issued ALESERV ADD for the hiperspace

The rules for using data-in-virtual and HSPSERV with the HSPALET parameter (for additional performance) are as follows:

- Your program can use HSPSERV with the HSPALET parameter with non-shared hiperspaces when a data-in-virtual object is mapped to a hiperspace, providing a DIV SAVE is not in effect.
- Once any program issues ALESERV ADD for a hiperspace, that hiperspace cannot be a data-in-virtual object.
- If a program issues ALESERV ADD for a hiperspace that is currently a data object, the system rejects the request.

For information on the use of ALETs with hiperspaces, see "Obtaining Additional HSPSERV Performance" on page 13-31.

The following two sections describe how your program can use the data-in-virtual services with hiperspaces.

Mapping a Data-in-Virtual Object to a Hiperspace

Through data-in-virtual, a program can map a VSAM linear data set residing on DASD to a hiperspace. The program uses the read and write operations of the HSPSERV macro to transfer data between the address space buffer area and the hiperspace window.

When a program maps a data-in-virtual object to a standard hiperspace, the system does not bring the data physically into the hiperspace; it reads the data into the address space buffer when the program uses HSPSERV SREAD for that area that contains the data.

Your program can map a single data-in-virtual object to several hiperspaces. Or, it can map several data-in-virtual objects to one hiperspace.

An Example of Mapping a Data-in-Virtual Object to a Hiperspace: The following example shows how you would create a standard hiperspace with a maximum size of one gigabyte and an initial size of 4K bytes. Figure 13-12 shows the hiperspace with a window that begins at the origin of the hiperspace.

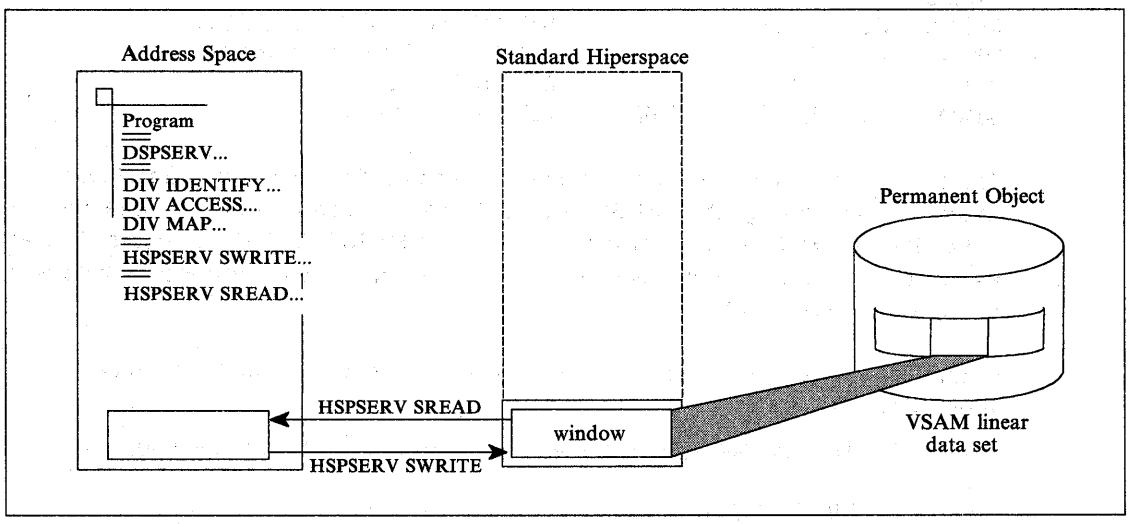


Figure 13-12. Example of Mapping a Data-in-Virtual Object to a Hiperspace

Initially, the window in the hiperspace and the buffer area in the address space are both 4K bytes. (That is, the window takes up the entire initial size of the hiperspace.) The data-in-virtual object is a VSAM linear data set on DASD.

```
* CREATE A STANDARD HIPERSPACE
  .
  DSPSERV CREATE,TYPE=HIPERSPACE,HSTYPE=SCROLL,NAME=HS1NAME,          X
     STOKEN=HS1STOK,BLOCKS=(ONEGIG,FOURK),ORIGIN=HS1ORG
  .
```

```
* MAP THE HIPERSPACE TO THE OBJECT
  .
  DIV     IDENTIFY,ID=OBJID,TYPE=DA,DDNAME=OBJDD
  DIV     ACCESS,ID=OBJID,MODE=UPDATE
  DIV     MAP,ID=OBJID,AREA=HS1ORG,STOKEN=HS1STOK
  .
* OBTAIN A 4K BUFFER AREA IN ADDRESS SPACE TO BE
* USED TO UPDATE THE DATA IN THE HIPERSPACE WINDOW
  .
*  DECLARATION STATEMENTS
  .
HS1NAME  DC   CL8'MYHSNAME'        HIPERSPACE NAME
HS1STOK  DS   CL8                  HIPERSPACE STOKEN
HS1ORG   DS   F                    HIPERSPACE ORIGIN
ONEGIG   DC   F'262144'            MAXIMUM SIZE OF 1G IN BLOCKS
FOURK    DC   F'1'                 INITIAL SIZE OF 4K IN BLOCKS
OBJID    DS   CL8                  DIV OBJECT ID
OBJDD    DC   AL1(7),CL7'MYDD   '  DIV OBJECT DDNAME
```

The program can read the data in the hiperspace window to a buffer area in the address space through the HSPSERV SREAD macro. It can update the data and write changes back to the hiperspace through the HSPSERV SWRITE macro. For an example of these operations, see "Example of Creating a Standard Hiperspace and Using It" on page 13-35.

Continuing the example, the following code saves the data in the hiperspace window on DASD and terminates the mapping.

```
* SAVE THE DATA IN THE HIPERSPACE WINDOW ON DASD AND END THE MAPPING
  .
  DIV     SAVE,ID=OBJID
  DIV     UNMAP,ID=OBJID,AREA=HS1ORG
  DIV     UNACCESS,ID=OBJID
  DIV     UNIDENTIFY,ID=OBJID
  .
* PROGRAM FINISHES USING THE DATA IN THE HIPERSPACE
  .
* DELETE THE HIPERSPACE
  .
  DSPSERV DELETE,STOKEN=HS1STOK
  .
```

Using a Hiperspace as a Data-in-Virtual Object

Your program can identify a non-shared standard hiperspace as a temporary data-in-virtual object, providing the hiperspace has never been the target of an ALESERV ADD. In this case, the window must be in an address space. Use the hiperspace for temporary storage of data, such as intermediate results of a computation. The movement of data between the window in the address space and the hiperspace object is through the DIV MAP and DIV SAVE macros. The data in the hiperspace is temporary.

Figure 13-13 shows an example of a hiperspace as a data-in-virtual object.

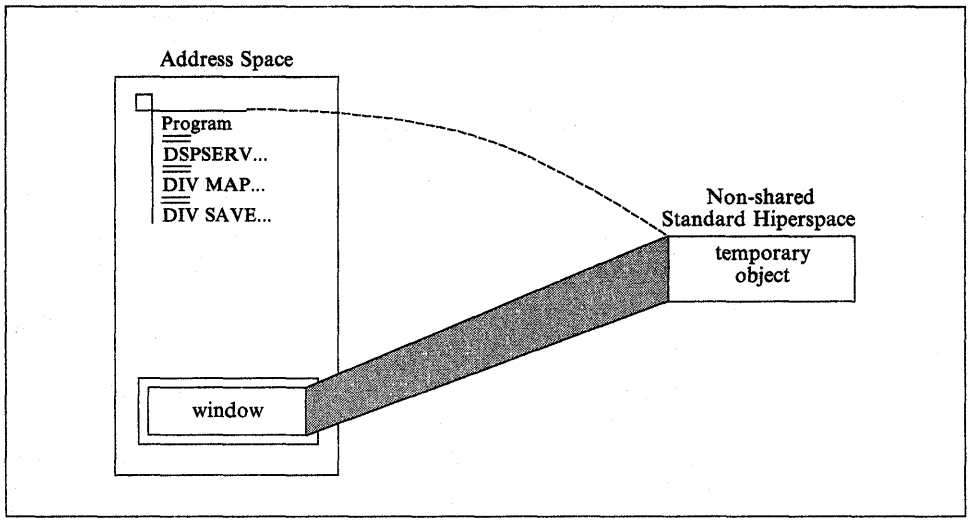


Figure 13-13. A Standard Hiperspace as a Data-in-Virtual Object

When the hiperspace is a data-in-virtual object, data-in-virtual services transfer data between the hiperspace object and the address space window. In this case, your program does not need to use, and must not use, HSPSERV SREAD and HSPSERV SWRITE.

An Example of a Hiperspace as a Data-in-Virtual Object: The program in this section creates a hiperspace for temporary storage of a table of 4K bytes that the program generates and uses. The program cannot save this table permanently.

The following code creates a standard hiperspace and identifies it as a data-in-virtual object.

```
* CREATE A HIPERSPACE
  .
  DSPSERV CREATE,TYPE=HIPERSPACE,HSTYPE=SCROLL,                    X
     NAME=HS2NAME,STOKEN=HS2STOK,BLOCKS=ONEBLOCK
  .
* IDENTIFY THE HIPERSPACE AS A DATA-IN-VIRTUAL OBJECT
  .
  DIV      IDENTIFY,ID=OBJID,TYPE=HS,STOKEN=HS2STOK
  DIV      ACCESS,ID=OBJID,MODE=UPDATE
  DIV      MAP,ID=OBJID,AREA=OBJAREA
  .
HS2NAME  DC   CL8'MHSNAME '        HIPERSPACE NAME
HS2STOK  DS   CL8                  HIPERSPACE STOKEN
ONEBLOCK DC   F'1'                 HIPERSPACE SIZE OF 1 BLOCK
OBJID    DS   CL8                  DIV OBJECT ID
OBJAREA  DS   CL8                  WINDOW IN ADDRESS SPACE
```

When the hiperspace is a data-in-virtual object, your program does not need to know the origin of the hiperspace. All addresses refer to offsets within the hiperspace. Note that the example does not include the ORIGIN parameter on DSPSERV.

After you finish making changes to the data in the address space window, you can save the changes back to the hiperspace as follows:

```
* SAVE CHANGES TO THE OBJECT
  .
  DIV      SAVE,ID=OBJID
```

The following macro refreshes the address space window. This means that if you make changes in the window and want a fresh copy of the object (that is, the copy that was saved last with the DIV SAVE macro), you would issue the following:

```
  DIV      RESET,ID=OBJID
```

When you finish using the hiperspace, use the DSPSERV macro to delete the hiperspace.

```
* DELETE THE HIPERSPACE
  .
  DSPSERV DELETE,STOKEN=HS2STOK
```

Chapter 14. Window Services

Callable window services enables assembler language programs to use the CALL macro to access data objects. By calling the appropriate window services program, an assembler language program can:

- Read or update an existing permanent data object
- Create and save a new permanent data object
- Create and use a temporary data object

Window services enable your program to access data objects without your program performing any input or output (I/O) operations. All your program needs to do is issue a CALL to the appropriate service program. The service performs any I/O operations that are required to make the data object available to your program. When you want to update or save a data object, window services again performs any required I/O operations.

Permanent Data Objects

A permanent data object is a virtual storage access method (VSAM) linear data set that resides on DASD. (This type of data set is also called a data-in-virtual object.) You can read data from an existing permanent object and also update the content of the object. You can create a new permanent object and when you are finished, save it on DASD. Because you can save this type of object on DASD, window services calls it a permanent object. Window services can handle very large permanent objects that contain as many as four gigabytes (4294967296 bytes).

Note: Installations whose high level language programs, such as FORTRAN, used data-in-virtual objects prior to MVS/SP 3.1.0 had to write an Assembler language interface program to allow the FORTRAN program to invoke the data-in-virtual program. Window services eliminates the need for this interface program.

Temporary Data Objects

A temporary data object is an area of expanded storage that window services provides for your program. You can use this storage to hold temporary data, such as intermediate results of a computation, instead of using a DASD workfile. Or you might use the storage area as a temporary buffer for data that your program generates or obtains from some other source. When you finish using the storage area, window services deletes it. Because you cannot save the storage area, window services calls it a temporary object. Window services can handle very large temporary objects that contain as many as 16 terabytes (17592186044416 bytes).

Structure of a Data Object

Think of a data object as a contiguous string of bytes organized into blocks, each 4096 bytes long. The first block contains bytes 0 to 4095 of the object, the second block contains bytes 4096 to 8191, and so forth.

Your program references data in the object by identifying the block or blocks that contain the desired data. Window services makes the blocks available to your program by mapping a window in your program storage to the blocks. A window is a storage area that your program provides and makes known to window services. Mapping the window to the blocks means that window services makes the data from those blocks available in the window when you reference the data. You can map a window to all or part of a data object depending on the size of the object and the size of the window. You can examine or change data that is in the window by using

the same instructions that you use to examine or change any other data in your program storage.

The following figure shows the structure of a data object and shows a window mapped to two of the object's blocks.

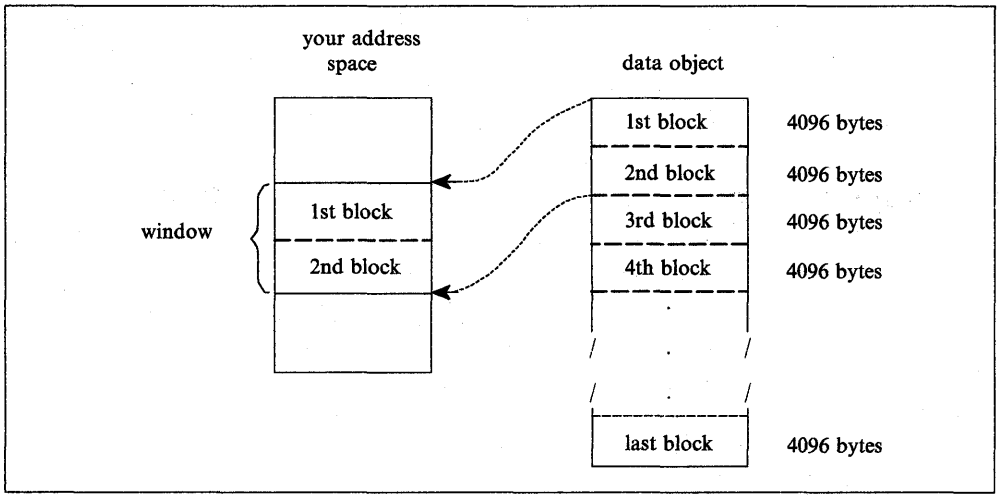


Figure 14-1. Structure of a Data Object

What Does Window Services Provide?

Window services allows you to view and manipulate data objects in a number of ways. You can have access to one or more data objects at the same time. You can also define multiple windows for a given data object. You can then view a different part of the object through each window. Before you can access any data object, you must request access from window services.

When you request access to a permanent data object, you must indicate whether you want a scroll area. A **scroll area** is an area of expanded storage that window services obtains and maps to the permanent data object. You can think of the permanent object as being available in the scroll area. When you request a view of the object, window services maps the window to the scroll area. If you do not request a scroll area, window services maps the window directly to the object on DASD.

A scroll area enables you to save interim changes to a permanent object without changing the object on DASD. Also, when your program accesses a permanent object through a scroll area, your program might attain better performance than it would if the object were accessed directly on DASD.

When you request a temporary object, window services provides an area of expanded storage. This area of expanded storage is the temporary data object. When you request a view of the object, window services maps the window to the temporary object. Window services initializes a temporary object to binary zeroes.

Notes:

1. Window services does not transfer data from the object on DASD, from the scroll area, or from the temporary object until your program references the data. Then window services transfers the blocks that contain the data your program requests.
2. The expanded storage that window services uses for a scroll area or for a temporary object is called a hiperspace. A hiperspace is a range of contiguous virtual storage addresses that a program can use like a buffer. Window services uses as many hiperspaces as needed to contain the data object.

The Ways That Window Services Can Map an Object

Window services can map a data object a number of ways. The following examples show how window services can:

- Map a permanent object that has no scroll area
- Map a permanent object that has a scroll area
- Map a temporary object
- Map an object to multiple windows
- Map multiple objects

Example 1 — Mapping a Permanent Object that has no Scroll Area

If a permanent object has no scroll area, window services maps the object from DASD directly to your window. In this example, your window provides a view of the first and second blocks of an object.

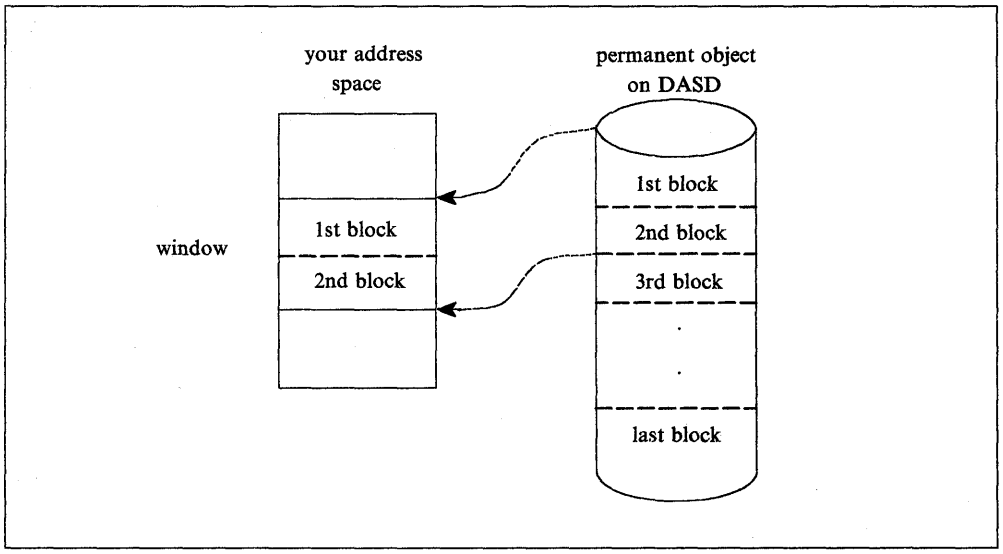


Figure 14-2. Mapping a Permanent Object That Has No Scroll Area

Example 2 — Mapping a Permanent Object that has a Scroll Area

If the object has a scroll area, window services maps the object from DASD to the scroll area. Window services then maps the blocks that you wish to view from the scroll area to your window. In this example, your window provides a view of the third and fourth blocks of an object.

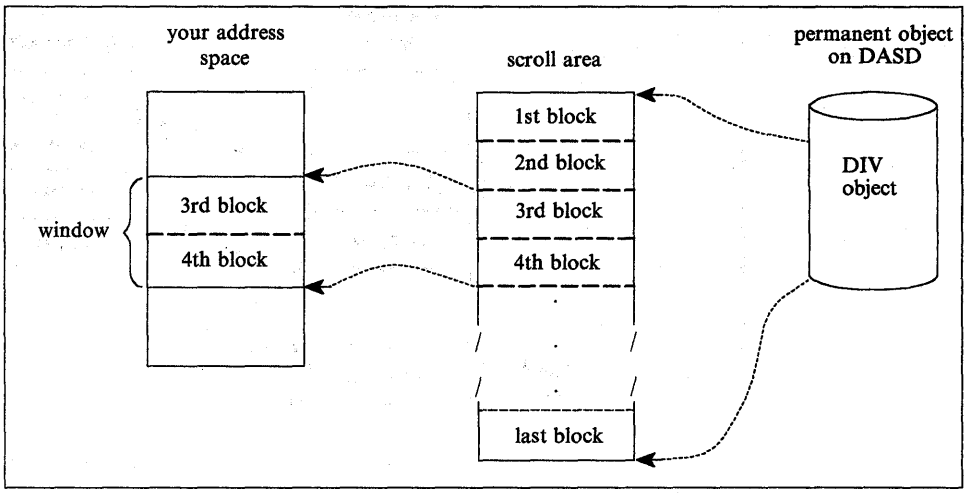


Figure 14-3. Mapping a Permanent Object That Has A Scroll Area

Example 3 — Mapping a Temporary Object

Window services uses a hiperspace as a temporary object. In this example, your window provides a view of the first and second blocks of a temporary object.

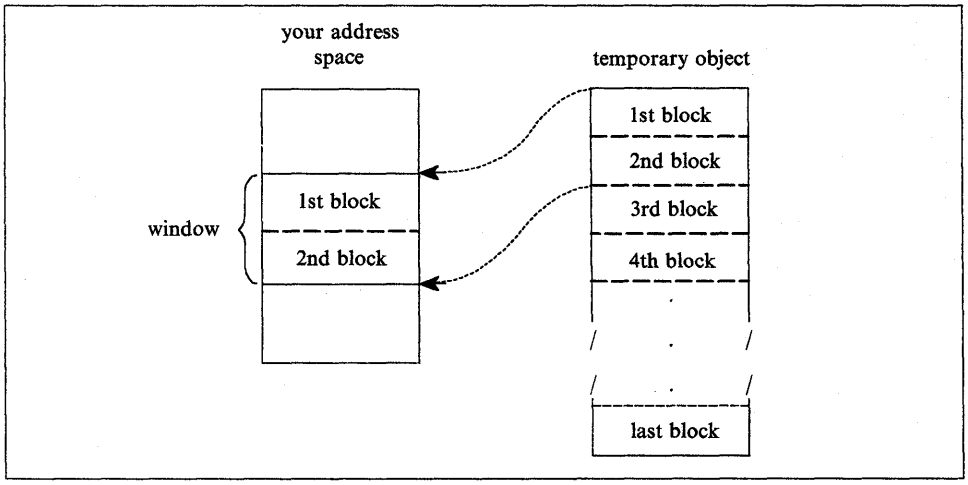


Figure 14-4. Mapping a Temporary Object

Example 4 — Mapping Multiple Windows to an Object

Window services can map multiple windows to the same object. In this example, one window provides a view of the second and third blocks of an object, and a second window provides a view of the last block.

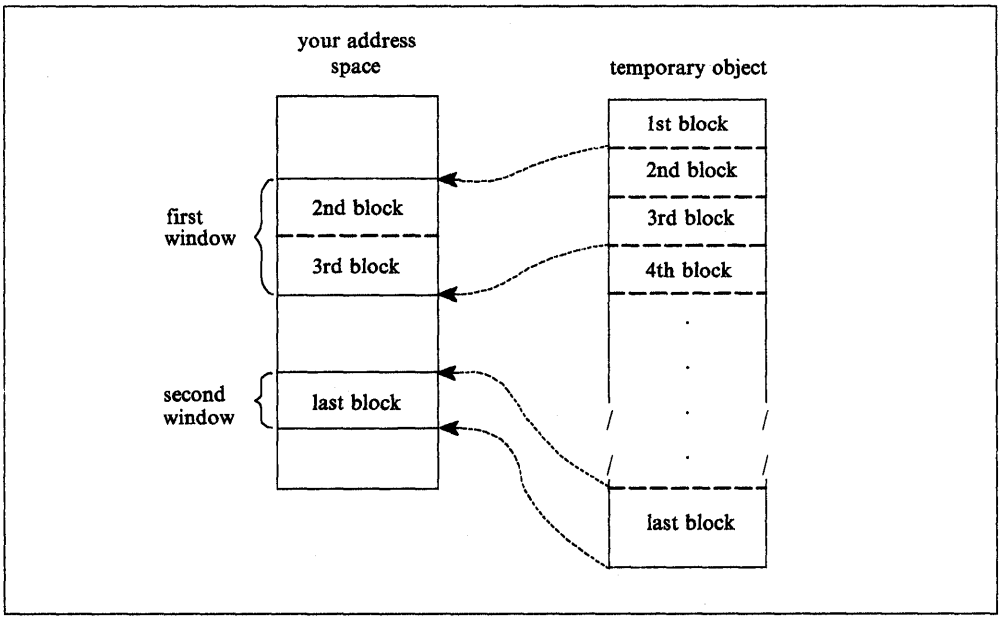


Figure 14-5. Mapping an Object To Multiple Windows

Example 5 — Mapping Multiple Objects

Window services can map multiple objects to windows in the same address space. The objects can be temporary objects, permanent objects, or a combination of temporary and permanent objects. In this example, one window provides a view of the second block of a temporary object, and a second window provides a view of the fourth and fifth blocks of a permanent object.

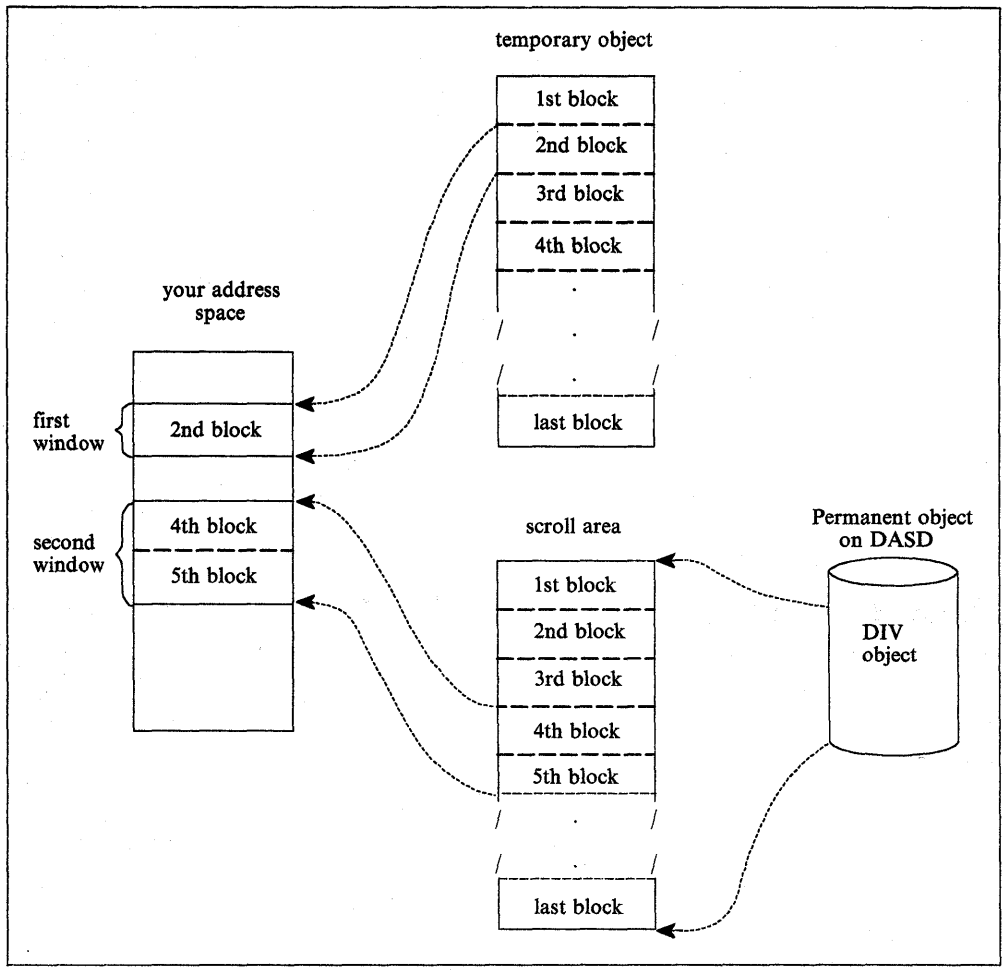


Figure 14-6. Mapping Multiple Objects

Access to Permanent Data Objects

When you have access to a permanent data object, you can:

- **View the object through one or more windows** — Depending on the object size and the window size, a single window can view all or part of a permanent object. If you define multiple windows, each window can view a different part of the object. For example, one window might view the first block of the permanent object and another window might view the second block. You can also have several windows view the same part of the object or have views in multiple windows overlap. For example, one window might view the first and second blocks of a data object while another window views the second and third blocks.
- **Change data that appears in a window** — You can examine or change data that is in a window by using the same instructions you use to examine or change any

other data in your program's storage. These changes do not alter the object on DASD or in the scroll area.

- **Save interim changes in a scroll area** — After changing data in a window, you can have window services save the changed blocks in a scroll area, if you have requested one. Window services replaces blocks in the scroll area with corresponding changed blocks from the window. Saving changes in the scroll area does not alter the object on DASD or alter data in the window.
- **Refresh a window or the scroll area** — After you change data in a window or save changes in the scroll area, you may discover that you no longer need those changes. In that case, you can have window services refresh the changed data. To refresh the window or the scroll area, window services replaces changed data with data from the object as it appears on DASD.
- **Replace the view in a window** — After you finish using data that's in a window, you can have window services replace the view in the window with a different view of the object. For example, if you are viewing the third, fourth, and fifth blocks of an object and are finished with those blocks, you might have window services replace that view with a view of the sixth, seventh, and eighth blocks.
- **Update the object on DASD** — If you have changes available in a window or in the scroll area, you can save the changes on DASD. Window services replaces blocks on DASD with corresponding changed blocks from the window and the scroll area. Updating an object on DASD does not alter data in the window or in the scroll area.

Access to Temporary Data Objects

When you have access to a temporary data object, you can:

- **View the object through one or more windows** — Depending on the object size and the window size, a single window can view all or part of a temporary object. If you define multiple windows, each window can view a different part of the object. For example, one window might view the first block of the temporary object and another window might view the second block. Unlike a permanent object, however, you cannot define multiple windows that have overlapping views of a temporary object.
- **Change data that appears in a window** — This function is the same for a temporary object as it is for a permanent object: you can examine or change data that is in a window by using the same instructions you use to examine or change any other data in your address space. These changes do not alter the object on DASD or in the scroll area.
- **Update the temporary object** — After you have changed data in a window, you can have window services update the object with those changes. Window services replaces blocks in the object with corresponding changed blocks from the window. The data in the window remains as it was.
- **Refresh a window or the object** — After you change data in a window or save changes in the object, you may discover that you no longer need those changes. In that case, you can have window services refresh the changed data. To refresh the window or the object, window services replaces changed data with binary zeroes.
- **Change the view in a window** — After you finish using data that's in a window, you can have window services replace the view in the window with a different view of the object. For example, if you are viewing the third, fourth, and fifth

blocks of an object and are finished with those blocks, you might have window services replace that view with a view of the sixth, seventh, and eighth blocks.

Using Window Services

To use, create, or update a data object, you call a series of programs that window services provides. These programs enable you to:

- Access an existing object, create and save a new permanent object, or create a temporary object
- Obtain a scroll area where you can make interim changes to a permanent object
- Define windows and establish views of an object in those windows
- Change or terminate the view in a window
- Update a scroll area or a temporary object with changes you have made in a window
- Refresh changes that you no longer need in a window or a scroll area
- Update a permanent object on DASD with changes that are in a window or a scroll area
- Terminate access to an object

The window services programs that you call and the sequence in which you call them depends on your use of the data object. For descriptions of the window services, see *Assembler Programming Reference*. For an example of invoking window services from an assembler language program, see "Window Services Coding Example" on page 15-15.

The first step in using any data object is to gain access to the object. To gain access, you call CSRIDAC. The object can be an existing permanent object, or a new permanent or temporary object you want to create. For a permanent object, you can request an optional scroll area. A scroll area enables you to make interim changes to an object's data without affecting the data on DASD. When CSRIDAC grants access, it provides an object identifier that identifies the object. You use that identifier to identify the object when you request other services from window service programs.

After obtaining access to an object, you must define one or more windows and establish views of the object in those windows. To establish a view of an object, you tell window services which blocks you want to view and in which windows. You can view multiple objects and multiple parts of each object at the same time. To define windows and establish views, you call CSRVIEW or CSREVW. After establishing a view, you can examine or change data that is in the window using the same instructions you use to examine or change other data in your program's storage.

After making changes to the part of an object that is in a window, you will probably want to save those changes. How you save changes depends on whether the object is permanent, is temporary, or has a scroll area.

If the object is permanent and has a scroll area, you can save changes in the scroll area without affecting the object on DASD. Later, you can update the object on DASD with changes saved in the scroll area. If the object is permanent and has no scroll area, you can update it on DASD with changes that are in a window. If the object is temporary, you can update it with changes that are in a window. To update an object on DASD, you call CSRSAVE. To update a temporary object or a scroll area, you call CSRSCOT.

After making changes in a window and possibly saving them in a scroll area or using them to update a temporary object, you might decide that you no longer need those changes. In this case, you can refresh the changed blocks. After refreshing a block of a permanent object or a scroll area to which a window is mapped, the refreshed block contains the same data that the corresponding block contains on DASD. After refreshing a block of a temporary object to which a window is mapped, the block contains binary zeroes. To refresh a changed block, you call CSRREFR.

After finishing with a view in a window, you can use the same window to view a different part of the object or to view a different object. Before changing the view in a window, you must terminate the current view. If you plan to view a different part of the same object, you terminate the current view by calling CSRVIEW. If you plan to view a different object or will not reuse the window, you can terminate the view by calling CSRIDAC.

When you finishing using a data object, you must terminate access to the object by calling CSRIDAC.

Obtaining Access to a Data Object

To obtain access to a permanent or temporary data object, call CSRIDAC. Indicate that you want to access an object, by specifying BEGIN as the value for *op_type*.

Identifying the Object

You must identify the data object you wish to access. How you identify the object depends on whether the object is permanent or temporary.

Permanent Object: For a permanent object, *object_name* and *object_type* work together. For *object_name* you have a choice: specify either the data set name of the object or the DDNAME to which the object is allocated. The *object_type* parameter must then indicate whether *object_name* is a DDNAME or a data set name:

- If *object_name* is a DDNAME, specify DDNAME as the value for *object_type*.
- If *object_name* is a data set name, specify DSNAME as the value for *object_type*.

If you specify DSNAME for *object_type*, indicate whether the object already exists or whether window services is to create it:

- If the object already exists, specify OLD as the value for *object_state*.
- If window services is to create the object, specify NEW as the value for *object_state*.

Requirement for NEW objects

If you specify NEW as the value for *object_state*, your system must include MVS/Data Facility Product (DFP) 3.1.0 and SMS must be active.

Temporary Object: To identify a temporary object, specify TEMPSPACE as the value for *object_type*. Window services assumes that a temporary object is new and must be created. Therefore, window services ignores the value assigned to *object_state*.

Specifying the Object's Size

If the object is permanent and new or is temporary, you must tell window services the size of the object. You specify object size through the *object_size* parameter. The size specified becomes the maximum size that window services will allow for that object. You express the size as the number of 4096-byte blocks needed to contain the object. If the number of blocks needed to contain the object is not an exact multiple of 4096, round *object_size* to the next whole number. For example:

- If the object size is to be less than 4097 bytes, specify 1.
- If the object size is 5000 bytes, specify 2.
- If the object size is 410,000 bytes, specify 101.

Specifying the Type of Access

For an existing (OLD) permanent object you must specify how you intend to access the object. You specify your intentions through the *access_mode* parameter:

- If you intend to only read the object, specify READ for *access_mode*.
- If you intend to update the object, specify UPDATE for *access_mode*.

For a new permanent object and for a temporary object, window services assumes you will update the object. In these cases, window services ignores the value assigned to *access_mode*.

Obtaining a Scroll Area

A scroll area is storage that window services provides for your use. This storage is outside your program's storage area and is accessible only through window services.

For a permanent object, a scroll area is optional. A scroll area allows you to make interim changes to a permanent object without altering the object on DASD. Later, if you want, you can update the object on DASD with the interim changes. A scroll area might also improve performance when your program accesses a permanent object.

For a temporary object, the scroll area is the object. Therefore, for a temporary object, a scroll area is required.

To indicate whether you want a scroll area, provide the appropriate value for *scroll_area*:

- To request a scroll area, supply a value of YES. YES is required for a temporary object.
- To indicate you do not want a scroll area, supply a value of NO.

Defining a View of a Data Object

To view all or part of a data object, you must provide window services with information about the object and how you want to view it. You must provide window services with the following information:

- The object identifier
- Where the window is in your address space
- Window disposition — that is, whether window services is to initialize the window the first time you reference data in the window
- Whether you intend to reference blocks of data sequentially or randomly
- The blocks of data that you want to view
- Whether you want to extend the size of the object

To define a view of a data object, call CSRVIEW or CSREVW. To determine which service you should use, see "Defining the Expected Reference Pattern" on page 14-12. Specify BEGIN as the value for *operation_type*.

Identifying the Data Object

To identify the object you want to view, specify the object identifier as the value for *object_id*. Use the same value CSRIDAC returned in *object_id* when you requested access to the object.

Identifying a Window

You must identify the window through which you will view the object. The window is a virtual storage area in your address space. You are responsible for obtaining the storage, which must meet the following requirements:

- The storage must not be page fixed.
- Pages in the window must not be page loaded (must not be loaded by the PGLOAD macro).
- The storage must start on a 4096 byte boundary and must be a multiple of 4096 bytes in length.

To identify the window, use the *window_name* parameter. The value supplied for *window_name* must be the symbolic name you assigned to the window storage area in your program.

Defining a window in this way provides one window through which you can view the object. To define multiple windows that provide simultaneous views of different parts of the object, see "Defining Multiple Views of an Object" on page 14-14.

Defining the Disposition of a Window's Contents

You must specify whether window services it is to replace or retain the window contents. You do this by selecting either the replace or retain option. This option determines how window services handles the data that is in the window the first time you reference the data. You select the option by supplying a value of REPLACE or RETAIN for *disposition*.

Replace Option: If you specify the replace option, the first time you reference a block to which a window is mapped, window services replaces the data in the window with corresponding data from the object. For example, assume you have requested a view of the first block of a permanent object and have specified the replace option. The first time you reference the window, window services replaces the data in the window with the first 4096 bytes (the first block) from the object.

If you've selected the replace option and then call CSRSAVE to update a permanent object, or call CSRSCOT to update a scroll area, or call CSRSCOT to update a temporary object, window services updates only the specified blocks that have changed and to which a window is mapped.

Select the replace option when you want to examine, use, or change data that's currently in an object.

Retain Option: If you select the retain option, window services retains data that is in the window. When you reference a block in the window the first time, the block contains the same data it contained before the reference.

When you select the retain option, window services considers all of the data in the window as changed. Therefore, if you call CSRSCOT to update a scroll area or a

temporary object, or call CSRSAVE to update a permanent object, window services updates all of the specified blocks to which a window or scroll area are mapped.

Select the retain option when you want to replace data in an object without regard for the data that it currently contains. You also use the retain option when you want to initialize a new object.

Defining the Expected Reference Pattern

You must tell window services whether you intend to reference the blocks of an object sequentially or randomly. An intention to access randomly tells window services to transfer one block (4096 bytes) of data into the window at a time. An intention to access sequentially tells window services to transfer more than one block into your window at one time. The performance gain is in having blocks of data already in central storage at the time the program needs to reference them. You specify the intent on either CSRVIEW or CSREVW, two services that differ on how to specify sequential access.

- CSRVIEW allows you a choice between random or sequential access.

  If you specify **RANDOM**, when you reference data that is not in your window, window services brings in one block — the one that contains the data your program references.

  If you specify **SEQ** for sequential, when you reference data that is not in your window, window services brings in up to 16 blocks — the one that contains the data your program requests, plus the next 15 consecutive blocks. The number of consecutive blocks varies, depending on the size of the window and availability of central storage. Use CSRVIEW if you are going to do one of the following:

  - Access randomly
  - Access sequentially, and you are satisfied with a maximum of 16 blocks coming into the window at a time.

- CSREVW is for sequential access only. It allows you to specify the maximum number of consecutive blocks that window services brings into the window at one time. The number ranges from one block through 256 blocks. Use CSREVW if you want fewer than 16 blocks or more than 16 blocks at one time. Programs that benefit from having more than 16 blocks come into a window at one time reference arrays that are greater than one megabyte. Often these programs perform significant amounts of numerically intensive computations.

To specify the reference pattern on CSRVIEW, supply a value of SEQ or RANDOM for *usage*.

To specify the reference pattern on CSREVW, supply a number from 0 through 255 for *pfcount*. *pfcount* represents the number of blocks window services will bring into the window, in addition to the one that it always brings in.

Note that window services brings in multiple pages differently depending on whether your object is permanent or temporary and whether the system has moved pages of your data from central storage to make those pages of central available for other programs. The rule is that SEQ on CSRVIEW and *pfcount* on CSREVW apply to:

- A **permanent object** when movement is from the object on DASD to central storage

- A **temporary object** when your program has scrolled the data out and references it again.

SEQ and *pfcount* do not apply after the system has moved data (either changed or unchanged) to auxiliary or expanded storage, and your program again references it, requiring the the system to bring the data back to central storage.

End the view whether established with CSRVIEW or CSREVW, with CSRVIEW END.

Identifying the Blocks You Want to View

To identify the blocks of data you want to view, use *offset* and *span*. The values you assign to *offset* and *span*, together, define a contiguous string of blocks that you want to view:

- The value assigned to *offset* specifies the relative block at which to start the view. An offset of 0 means the first block; an offset of 1 means the second block; an offset of 2 means the third block, and so forth.
- The value assigned to *span* specifies the number of blocks to view. A span of 1 means one block; a span of 2 means two blocks, and so forth. A span of 0 has special meaning: it means the view is to start at the specified offset and extend until the currently defined end of the object.

The following table shows examples of several *offset* and *span* combinations and the resulting view in the window.

Offset	Span	Resulting view in the window
0	0	view the entire object
0	1	view the first block only
1	0	view the second block through the last block
1	1	view the second block only
2	2	view the third and fourth blocks only

Extending the Size of a Data Object

You can use *offset* and *span* to extend the size of an object up to the previously defined maximum size for the object. You can extend the size of either permanent objects or temporary objects. For objects created through CSRIDAC, the value assigned to *object_size* defines the maximum allowable size. When you call CSRIDAC to gain access to an object, CSRIDAC returns a value in *high_offset* that defines the current size of the object.

For example, assume you have access to a permanent object whose maximum allowable size is four 4096-byte blocks. The object is currently two blocks long. If you define a window and specify an offset of 1 and a span of 2, the window contains a view of the second block and a view of a third block which does not yet exist in the permanent object. When you reference the window, the content of the second block, as seen in the window, depends on the disposition you selected, replace or retain. The third block, as seen in the window, initially contains binary zeroes. If you later call CSRSAVE to update the permanent object with changes from the window, window services extends the size of the permanent object to three blocks by appending the new block of data to the object.

Defining Multiple Views of an Object

You might need to view different parts of an object at the same time. For a permanent object, you can define windows that have non-overlapping views as well as windows that have overlapping views. For a temporary object, you can define windows that have only non-overlapping views.

- A non-overlapping view means that no two windows view the same block of the object. For example, a view is non-overlapping when one window views the first and second blocks of an object and another window views the ninth and tenth blocks of the same object. Neither window views a common block.
- An overlapping view means that two or more windows view the same block of the object. For example, the view overlaps when the second window in the previous example views the second and third blocks. Both windows view a common block, the second block.

Non-Overlapping Views

To define multiple windows that have a non-overlapping view, call CSRIDAC once to obtain the object identifier. Then call CSRVIEW or CSREVW once to define each window. On each call, specify BEGIN to define the type of operation, and specify the same object identifier for *object_id*, and a different value for *window_name*. Define each window's view by specifying values for *offset* and *span* that create windows with non-overlapping views.

Overlapping Views

To define multiple windows that have an overlapping view of a permanent object, define each window as though it were viewing a different object. That is, define each window under a different object identifier. To obtain the object identifiers, call CSRIDAC once for each identifier you need. Only one of the calls to CSRIDAC can specify an access mode of UPDATE. Other calls to CSRIDAC must specify an access mode of READ.

After calling CSRIDAC, call CSRVIEW or CSREVW once to define each window. On each call, specify BEGIN to define the operation, and specify a different object identifier for *object_id*, and a different value for *window_name*. Define each window's view by specifying values for *offset* and *span* that create windows with the required overlapping views.

To define multiple windows that have an overlapping view, define each window as though it were viewing a different object. That is, define each window under a different object identifier. To obtain the object identifiers, call CSRIDAC once for each identifier you need. Then call CSRVIEW or CSREVW once to define each window. On each call, specify the value BEGIN for the operation type, and specify a different object identifier for *object_id*, and a different value for *window_name*. Define each window's view by specifying values for *offset* and *span* that create windows with the required overlapping views.

Saving Interim Changes to a Permanent Data Object

Window services allows you to save interim changes you make to a permanent object. You must have previously requested a scroll area for the object, however. You request a scroll area when you call CSRIDAC to gain access to the object. Window services saves changes by replacing blocks in the scroll area with corresponding changed blocks from a window. Saving changes in the scroll area does not alter the object on DASD.

After you have a view of the object and have made changes in the window, you can save those changes in the scroll area. To save changes in the scroll area, call CSRSCOT. To identify the object, you must supply an object identifier for *object_id*. The value supplied for *object_id* must be the same value CSRIDAC returned in *object_id* when you requested access to the object.

To identify the blocks in the object that you want to update, use *offset* and *span*. The values assigned to *offset* and *span*, together, define a contiguous string of blocks in the object:

- The value assigned to *offset* specifies the relative block at which to start. An offset of 0 means the first block; an offset of 1 means the second block; an offset of 2 means the third block, and so forth.
- The value assigned to *span* specifies the number of blocks to save. A span of 1 means one block; a span of 2 means two blocks, and so forth. A span of 0 has special meaning: it requests that window services save all changed blocks to which a window is mapped.

Window services replaces each block within the range specified by *offset* and *span* providing the block has changed and a window is mapped to the block.

Updating a Temporary Data Object

After making changes in a window to a temporary object, you can update the object with those changes. You must identify the object and must specify the range of blocks that you want to update. To be updated, a block must be mapped to a window and must contain changes in the window. Window services replaces each block within the specified range with the corresponding changed block from a window.

To update a temporary object, call CSRSCOT. To identify the object, you must supply an object identifier for *object_id*. The value you supply for *object_id* must be the same value CSRIDAC returned in *object_id* when you requested access to the object.

To identify the blocks in the object that you want to update, use *offset* and *span*. The values assigned to *offset* and *span*, together, define a contiguous string of blocks in the object:

- The value assigned to *offset* specifies the relative block at which to start. An offset of 0 means the first block; a offset of 1 means the second block; an offset of 2 means the third block, and so forth.
- The value assigned to *span* specifies the number of blocks to save. A span of 1 means one block; a span of 2 means two blocks, and so forth. A span of 0 has special meaning: it requests that window services update all changed blocks to which a window is mapped.

Window services replaces each block within the range specified by *offset* and *span* providing the block has changed and a window is mapped to the block.

Refreshing Changed Data

You can refresh blocks that are mapped to either a temporary object or to a permanent object. You must identify the object and specify the range of blocks you want to refresh. When you refresh blocks mapped to a temporary object, window services replaces, with binary zeros, all changed blocks that are mapped to the window. When you refresh blocks mapped to a permanent object, window services

replaces specified changed blocks in a window or in the scroll area with corresponding blocks from the object on DASD.

To refresh an object, call CSRREFR. To identify the object, you must supply an object identifier for *object_id*. The value supplied for *object_id* must be the same value CSRIDAC returned in *object_id* when you requested access to the object.

To identify the blocks of the object that you want to refresh, use *offset* and *span*. The values assigned to *offset* and *span*, together, define a contiguous string of blocks in the object:

- The value assigned to *offset* specifies the relative block at which to start. An offset of 0 means the first block; an offset of 1 means the second block; an offset of 2 means the third block, and so forth.
- The value assigned to *span* specifies the number of blocks to save. A span of 1 means one block; a span of 2 means two blocks, and so forth. A span of 0 has special meaning: it requests that window services refresh all changed blocks to which a window is mapped, or refresh all changed blocks that have been saved in a scroll area.

Window services refreshes each block within the range specified by *offset* and *span* providing the block has changed and a window or a scroll area is mapped to the block. At the completion of the refresh operation, blocks from a permanent object that have been refreshed appear the same as the corresponding blocks on DASD. Refreshed blocks from a temporary object contain binary zeroes.

Updating a Permanent Object on DASD

You can update a permanent object on DASD with changes that appear in a window or in the object's scroll area. You must identify the object and specify the range of blocks that you want to update.

To update an object, call CSRSAVE. To identify the object, you must supply an object identifier for *object_id*. The value you provide for *object_id* must be the same value CSRIDAC returned when you requested access to the object.

To identify the blocks of the object that you want to update, use *offset* and *span*. The values assigned to *offset* and *span*, together, define a contiguous string of blocks in the object:

- The value assigned to *offset* specifies the relative block at which to start. An offset of 0 means the first block; an offset of 1 means the second block; an offset of 2 means the third block, and so forth.
- The value assigned to *span* specifies the number of blocks to save. A span of 1 means one block; a span of 2 means two blocks, and so forth. A span of 0 has special meaning: it requests that window services update all changed blocks to which a window is mapped, or update all changed blocks that have been saved in the scroll area.

When There is a Scroll Area

When the object has a scroll area, window services first updates blocks in the scroll area with corresponding blocks from windows. To be updated, a scroll area block must be within the specified range, a window must be mapped to the block, and the window must contain changes. Window services next updates blocks on DASD with corresponding blocks from the scroll area. To be updated, a DASD block must be within the specified range and have changes in the scroll area. Blocks in the window remain unchanged.

When There is No Scroll Area

When there is no scroll area, window services updates blocks of the object on DASD with corresponding blocks from a window. To be updated, a DASD block must be within the specified range, mapped to a window, and have changes in the window. Blocks in the window remain unchanged.

Changing a View in a Window

To change the view in a window so you can view a different part of the same object or view a different object, you must first terminate the current view. To terminate the view, whether the view was established by CSRVIEW or CSREVW, call CSRVIEW and supply a value of END for *operation_type*. You must also identify the object, identify the window, identify the blocks you are currently viewing, and specify a disposition for the data that is in the window.

To identify the object, supply an object identifier for *object_id*. The value supplied for *object_id* must be the value you supplied when you established the view.

To identify the window, supply the window name for *window_name*. The value supplied for *window_name* must be the same value you supplied when you established the view.

To identify the blocks you are currently viewing, supply values for *offset* and *span*. The values you supply must be the same values you supplied for *offset* and *span* when you established the view.

To specify a disposition for the data you are currently viewing, supply a value for *disposition*. The value determines what data will be in the window after the CALL to CSRVIEW completes.

- For a permanent object that has no scroll area:
  - To retain the data that's currently in the window, supply a value of RETAIN for *disposition*.
  - To discard the data that's currently in the window, supply a value of REPLACE for *disposition*. After the operation completes, the window contents are unpredictable.

  For example, assume that a window is mapped to one block of a permanent object that has no scroll area. The window contains the character string AAA.....A and the block to which the window is mapped contains BBB.....B. If you specify a value of RETAIN, upon completion of the CALL, the window still contains AAA.....A, and the mapped block contains BBB.....B. If you specify a value of REPLACE, upon completion of the CALL, the window contents are unpredictable and the mapped block still contains BBB.....B.
- For a permanent object that has a scroll area or for a temporary object:
  - To retain the data that's currently in the window, supply a value of RETAIN for *disposition*. CSRVIEW or CSREVW also updates the mapped blocks of the scroll area or temporary object so that they contain the same data as the window.
  - To discard the data that's currently in the window, supply a value of REPLACE for *disposition*. Upon completion of the operation, the window contents are unpredictable.

  For example, assume that a window is mapped to one block of a temporary object. The window contains the character string AAA.....A and the block to

which the window is mapped contains BBB.....B. If you specify a value of RETAIN, upon completion of the CALL, the window still contains AAA.....A and the mapped block of the object also contains AAA.....A. If you specify a value of REPLACE, upon completion of the CALL, the window contents are unpredictable and the mapped block still contains BBB.....B.

CSRVIEW ignores the values you assign to the other parameters.

When you terminate the view of an object, the type of object that is mapped and the value you specify for *disposition* determine whether CSRVIEW updates the mapped blocks. CSRVIEW updates the mapped blocks of a temporary object or a permanent object's scroll area if you specify a disposition of RETAIN. In all other cases, to update the mapped blocks, call the appropriate service before terminating the view:

- To update a temporary object, or to update the scroll area of a permanent object, call CSRSCOT.
- To update an object on DASD, call CSRSAVE.

Upon successful completion of the CSRVIEW operation, the content of the window depends on the value specified for disposition. The window is no longer mapped to a scroll area or to an object, however. The storage used for the window is available for other use, perhaps to use as a window for a different part of the same object or to use as a window for a different object.

Terminating Access to a Data Object

When you finish using a data object, you must terminate access to the object. When you terminate access, window services returns to the system any virtual storage it obtained for the object: storage for a temporary object or storage for a scroll area. If the object is temporary, window services deletes the object. If the object is permanent and window services dynamically allocated the data set when you requested access to the object, window services dynamically unallocates the data set. Your window is no longer mapped to the object or to a scroll area.

When you terminate access to a permanent object, window services does not update the object on DASD with changes that are in a window or the scroll area. To update the object, call CSRSAVE before terminating access to the object.

To terminate access to an object, call CSRIDAC and supply a value of END for *operation_type*. To identify the object, supply an object identifier for *object_id*. The value you supply for *object_id* must be the same value CSRIDAC returned when you obtained access to the object.

Upon successful completion of the call, the storage used for the window is available for other use, perhaps as a window for viewing a different part of the same object or to use as a window for viewing a different object.

Link-editing Callable Window Services

Any program that invokes window services must be link-edited with an IBM-provided linkage-assist routine. The linkage-assist routine provides the logic needed to locate and invoke the callable services. The linkage-assist routine resides in SYS1.CSSLIB. The following example shows the JCL needed to link-edit a program with the linkage-assist routine.

```
//LINKJOB    JOB  'accountinfo','name',CLASS=x,
//                MSGCLASS=x,NOTIFY=userid,MSGLEVEL=(1,1),REGION=4096K
//LINKSTP1   EXEC  PGM=HEWLH096,PARM='LIST,LET,XREF,REFR,RENT,NCAL,
//                SIZE=(1800K,128K)'
//SYSPRINT   DD    SYSOUT=x
//SYSLMOD    DD    DSNAME=userid.LOADLIB,DISP=SHR
//SYSUT1     DD    UNIT=SYSDA,SPACE=(TRK,(5,2))
//SYSLIN     DD    *
  INCLUDE    OBJDD1(userpgm)
  INCLUDE    OBJDD2(CSRCPOOL)
  NAME       userpgm(R)
//OBJDD1     DD    DSN=userid.OBJLIB,DISP=SHR
//OBJDD2     DD    DSN=SYS1.CSSLIB,DISP=SHR
```

The example JCL assumes that the program you are link-editing is reentrant.

Window Services Coding Example

This example shows the code needed to invoke window services from an assembler language program. Use this example to supplement and reinforce information that is presented elsewhere in this chapter.

```
EXAMPLE1 CSECT
         STM 14,12,12(13)          Save caller's registers in caller's
*                                  save area
         LR  12,15                 Set up R12 as the base register
         USING EXAMPLE1,12
*        .
*        .
*        .
*********************************************************************
*  Set up save area                                                 *
*********************************************************************
         LA  15,SAVEAREA           Load address of save area into R15
         ST  13,4(15)              Save address of caller's save area
*                                  into this program's save area
         ST  15,8(13)              Save address of this program's save
*                                  area into caller's save area
         LR  13,15                 Load address of save area into R13
*        .
*        .                         Program continues....
*        .
*********************************************************************
*  Call CSRIDAC to identify and access an old data object, request *
*  a scroll area, and get update access.                            *
*********************************************************************
         CALL CSRIDAC,(OPBEGIN,DDNAME,OBJNAME,YES,OLD,ACCMODE,        *
               OBJSIZE,OBJID1,LSIZE,RC,RSN)
*        .
*        .                         Program continues....
*        .
*********************************************************************
*  GETMAIN 50 pages of virtual storage to use as a window           *
*********************************************************************
         GETMAIN  RU,LV=GSIZE,BNDRY=PAGE,A=WINDWPTR    Getmain a
*                                  window of 50 pages
         L  R3,WINDWPTR            Move the address of the window into
*                                  register 3
         USING  WINDOW,R3          Sets up WINDOW as based off of reg 3
*********************************************************************
*  Call CSRVIEW to set up a map of 50 blocks between the getmained *
*  virtual storage and the data object.                             *
*********************************************************************
         LA R4,ZERO                LOAD A ZERO INTO REGISTER 4
         ST R4,OFFSET1             Initialize offset to 0 to indicate
*                                  the beginning of the data object
         CALL CSRVIEW,(OPBEGIN,OBJID1,OFFSET1,SPAN1,WINDOW,ACCSEQ,    *
```

```
              REPLACE,RC,RSN)
*         .
*         .                          Program continues....
*         .                          write data in the window
*         .
*********************************************************************
*  Call CSRSAVE to write data in the window to the first 50 blocks *
*  of the data object                                               *
*********************************************************************
          CALL CSRSAVE,(OBJID1,OFFSET1,SPAN1,LSIZE,RC,RSN)
*         .
*         .                          Program continues....
*         .                          change data in the window
*         .
*********************************************************************
*  Call CSRSCOT to write new data in the window to the first 50     *
*  blocks of the scroll area                                        *
*********************************************************************
          CALL CSRSCOT,(OBJID1,OFFSET1,SPAN1,RC,RSN)
*         .
*         .                          Program continues....
*         .                          change data in the window
*         .
*********************************************************************
*  Call CSRREFR to refresh the window, that is, get back the last   *
*  SAVEd data in the data object                                    *
*********************************************************************
          CALL CSRREFR,(OBJID1,OFFSET1,SPAN1,RC,RSN)
*         .
*         .                          Program continues....
*         .
*********************************************************************
*  Call CSRIDAC to unidentify and unaccess the data object          *
*********************************************************************
          CALL CSRIDAC,(OPEND,DDNAME,OBJNAME,YES,OLD,ACCMODE,           *
                OBJSIZE,OBJID1,LSIZE,RC,RSN)
*         .
*         .                          Program continues....
*         .
          BR 14                      End of EXAMPLE1
ZERO      EQU  0                     Constant zero
GSIZE     EQU  204800                Getmain a window of 50 pages (blocks)
          DS 0D
OPBEGIN   DC CL5'BEGIN'              Operation type BEGIN
OPEND     DC CL4'END '               Operation type END
DDNAME    DC CL7'DDNAME '            Object type DDNAME
OBJNAME   DC CL8'MYDDNAME'           DDNAME of data object
YES       DC CL3'YES'                Yes for a scroll area
OLD       DC CL3'OLD'                Data object already exists
ACCSEQ    DC CL4'SEQ '               Sequential access
ACCMODE   DC CL6'UPDATE'             Update mode
REPLACE   DC CL7'REPLACE'            Replace data in window on a map
OBJSIZE   DC F'524288'               Size of data object is 2 gig
SPAN1     DC F'50'                   Set up a span of 50 blocks
OBJID1    DS CL8                     Object identifier
LSIZE     DS F                       Logical size of data object
OFFSET1   DS F                       Offset into data object
RC        DS F                       Return code from service
RSN       DS F                       Reason code from service
SAVEAREA  DS 18F                     This program's save area
WINDWPTR  DS F                       Address of getmained window
WINDOW    DSECT                      Mapping of window to view the
          DS CL204800                object data
          END
```

Chapter 15. Processor Storage Management

The system administers the use of processor storage (that is, central and expanded storage) and it directs the movement of virtual pages between auxiliary, expanded, and central storage in page size (4096-byte or 4K-byte) blocks. It makes all addressable virtual storage in each address space and data space or hiperspace appear as central storage. Virtual pages necessary for program execution are kept in processor storage as long as:

- The program references the pages frequently enough
- Other programs do not need that same central storage.

The system performs the paging I/O necessary to transfer pages in and out of central storage and also provides DASD allocation and management for paging I/O space on auxiliary storage.

The system assigns real frames upon request from a pool of available real frames, thereby associating virtual addresses with real addresses. Frames are repossessed upon termination of use, when freed by a user, when a user is swapped-out, or when needed to replenish the available pool. While a virtual page occupies a real frame, the page is considered pageable unless specified otherwise as a system page that must be resident in central storage. The system also allocates virtual equals central (V=R) regions upon request by those programs that cannot tolerate dynamic relocation. Such a region is allocated contiguously from a predefined area of central storage and is non-pageable. Programs in this region do run in dynamic address translation (DAT) mode, although real and virtual addresses are equivalent.

This chapter describes how you can:

- Free the virtual storage in your address space and the virtual storage in any data space that you might have access to
  - FREEMAIN and STORAGE RELEASE frees specific portions of virtual storage in address spaces.
  - DSPSERV DELETE frees all of the virtual storage in a data space or hiperspace.
- Release the central and expanded storage that actually holds the data that your program has in virtual storage.
  - PGRLSE or PGSER RELEASE releases specified portions of virtual storage contents of an address space.
  - DSPSERV RELEASE releases specified portions of virtual storage contents of a data space or hiperspace.
- Request that the system pre-load or page out central storage
  - PGLOAD or PGSER LOAD loads specified virtual storage areas of an address space into central storage.
  - PGOUT or PGSER OUT pages out specified virtual storage areas of an address space from central storage.
  - DSPSERV LOAD loads specified virtual storage areas of a data space into central storage.
  - DSPSERV OUT pages out specified virtual storage areas of a data space from central storage.

- Request that the system preload multiple pages on a page fault.
  - REFPAT causes the system to preload pages according to a program's reference pattern. REFPAT is intended for numerically intensive programs.

Freeing Virtual Storage

All storage obtained for your program by GETMAIN, STORAGE OBTAIN, or DSPSERV CREATE is automatically freed by the system when the job step terminates. Freeing storage in this manner requires no action on your part.

FREEMAIN or STORAGE RELEASE perform the equivalent of a page release for any resulting free page and the page is no longer available to the issuer. DSPSERV DELETE performs the same action for a data space that FREEMAIN and STORAGE RELEASE do for address space virtual storage except that for a data space or hiperspace, **all** of the storage is released.

Releasing Storage

When your program is finished using an area of virtual storage, it can release the storage to make the central, expanded, or auxiliary storage that actually holds the data available for other uses. The decision to release the storage depends on the size of the storage and when the storage will be used again:

- For large areas (over 100 pages, for example) that will not be used for five or more seconds of processor time, consider releasing the storage. If you do not release those pages after you are finished using them:
  - Your program might be using central storage that could better be used for other purposes.
  - Your program might have delays later when the system moves your pages from central storage to expanded or auxiliary storage.
- Generally, for smaller amounts of storage that will be used again in five seconds or less, do not release the storage.

Note that **releasing** storage does not **free** the virtual storage.

When releasing storage for an address space, use PGRLSE or PGSER with the RELEASE parameter. As shown in Figure 15-1, if the specified addresses are not on page boundaries, the low address is rounded up and the high address is rounded down; then, the pages contained between the addresses are released.

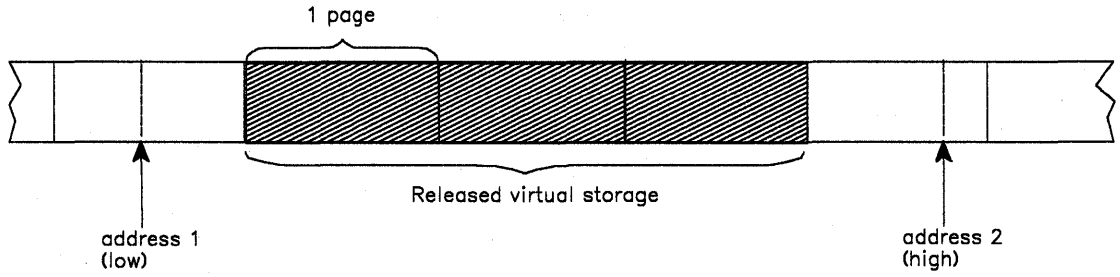


Figure 15-1. Releasing Virtual Storage

When releasing storage for a data space or hiperspace, use the DSPSERV RELEASE macro to release the central, expanded or auxiliary storage that actually holds the data. The starting address must be on a 4K-byte boundary and you can release data space storage only in increments of 4K bytes.

For both address spaces and data spaces, the virtual space remains, but its contents are discarded. When the using program can discard the contents of a large virtual area (one or more complete pages) and reuse the virtual space without the necessity of paging operations, the page release function may improve operating efficiency.

Loading/Paging Out Virtual Storage Areas

The PGLOAD, PGSER LOAD, and DSPSERV LOAD essentially provide a page-ahead function. By loading specified address space and data space areas into central storage, you can attempt to ensure that certain pages will be in central storage when needed. Page faults can still occur, however, because these pages may be paged out if not referenced soon enough.

Loading and paging for address spaces: With the page load function, you have the option of specifying that the contents of the virtual area is to remain intact or be released. If you specify RELEASE=Y with PGLOAD or PGSER LOAD, the current contents of entire virtual 4K pages to be brought in may be discarded and new real frames assigned without page-in operations; if you specify RELEASE=N, the contents are to remain intact and be used later. If you specify RELEASE=Y, the page release function will be performed before the page load function. That is, no page-in is needed for areas defining entire virtual pages since the contents of those pages are expendable.

Loading and paging for data spaces: DSPSERV LOAD requests the starting address of the data space area to be loaded and the number of pages that the system is to load. It does not offer a RELEASE=Y or a RELEASE=N function.

PGOUT, PGSER OUT, and DSPSERV OUT initiate page-out operations for specified virtual areas that are in central storage. For address spaces, the real frames will be made available for reuse upon completion of the page-out operation unless you specify the KEEPREL parameter in the macro. An area that does not encompass one or more complete pages will be copied to auxiliary storage, but the real frames will not be freed. DSPSERV LOAD does not have the KEEPREL function.

The proper use of the page load and page out functions tend to decrease system overhead by helping the system keep pages currently in use, or soon to be in use, in central storage. Improper use of the page load and page out functions can lead to excessive system overhead. An example of the misuse of the page load function is to load ten pages and then use only two.

For more information on DSPSERV LOAD and DSPSERV OUT, see "Paging Data Space Storage Areas into and out of Central Storage" on page 13-17.

Virtual Subarea List (VSL)

The virtual subarea list provides the basic input to the page service functions that use a 24-bit interface: PGLOAD, PGRLSE, and PGOUT. The list consists of one or more doubleword entries, each entry describing an area of virtual storage. The list must be nonpageable and contained in the address space of the subarea to be processed.

Each parameter list entry has the following format:

Byte	0	1	2	3	4	5	6	7
	FLAGS	START ADDRESS			FLAGS	END ADDRESS + 1		

Byte 0 Flags:

Bit 0	(1... ....)	This bit indicates that bytes 1-3 are a chain pointer to the next VSL entry to be processed; bytes 4-7 are ignored. This feature allows several parameter lists to be chained as a single logical parameter list.
Bit 1	(.1.. ....)	Reserved.
Bit 2	(..1. ....)	Reserved.
Bit 3	(...1 ....)	PGLOAD is to be performed; reserved, set by macro.
Bit 4	(.... 1...)	PGRLSE is to be performed; reserved, set by macro.
Bit 5	(.... .1..)	Reserved.
Bit 6	(.... ..1.)	Reserved.
Bit 7	(.... ...1)	Reserved.

Start Address:
The virtual address of the origin of the virtual area to be processed.

Byte 4 Flags:

Bit 0	(1... ....)	This flag indicates the last entry of the list. It is set in the last doubleword entry in the list.
Bit 1	(.1.. ....)	When this flag is set, the entry in which it is set is ignored.
Bit 2	(..1. ....)	Reserved.
Bit 3	(...1 ....)	This flag indicates that a return code of 4 was issued from a page service function other than PGRLSE.
Bit 4	(.... 1...)	Reserved.
Bit 5	(.... .1..)	PGOUT is to be performed; reserved, set by macro.
Bit 6	(.... ..1.)	KEEPREL option of PGOUT is to be performed; reserved, set by macro.
Bit 7	(.... ...1)	Reserved.

End Address + 1:
The virtual address of the byte immediately following the end of the virtual area.

Page Service List (PSL)

The page services list provides the basic input to the page service function for the PGSER macro. Specify 31-bit addresses in the PSL entries. Each PSL entry specifies the range of addresses for which a service is to be performed or points to the first PSL entry in a new list of concatenated PSL entries that are to be processed. Within a PSL entry, you can also nullify a service on a range of addresses by indicating that you do not want to perform the service for that range.

Each 12-byte PSL entry has the following form:

Bytes	Meaning
0-3	Bit 0 of byte 0 must be 0. The remainder of these bytes contains the 31-bit starting address for which the page service is to be performed or a pointer to the next PSL.
4-7	Bit 0 of byte 4 must be 0. If bytes 0-3 contain the starting address, these bytes contain the address of the last byte for which the page service is to be performed. If bytes 0-3 contain a pointer to the next PSL, these bytes are reserved.
8	Flags set by the caller as follows: **Bit Meaning** 0 Set to 1 to indicate that this is the last PSL entry in a concatenation of PSL entries. 1 Set to 1 to indicate that no services are to be performed for the range of addresses specified. 2 Set to 1 to indicate that bytes 0-3 contain a pointer to the next PSL.
9-11	Set by the PGSER service routine.

Defining the Reference Pattern (REFPAT)

The REFPAT macro allows a program to define a reference pattern for a specified area that the program is about to reference. Additionally, the program specifies how much data it wants the system to attempt to bring into central storage on a page fault. The system honors the request according to the availability of central storage. By bringing in more data at a time, the system takes fewer page faults; fewer page faults means possible improvement in performance.

Programs that benefit from REFPAT are those that reference amounts of data that are greater than one megabyte. The program should reference the data in a sequential manner, either forward or backward. In addition, if the program "skips over" certain areas, and these areas are of uniform size and are repeated at regular intervals, REFPAT might provide additional performance improvement. Although REFPAT affects movement of pages from auxiliary **and** expanded storage, the greatest gain is for movement of pages from auxiliary storage.

There are two REFPAT services:

- REFPAT INSTALL identifies the data area and the reference pattern, and specifies the number of bytes that the system is to try to bring into central storage at one time. These activities are called "defining the reference pattern."

- REFPAT REMOVE removes the definition; it tells the system that the program has stopped using the reference pattern for the specified data area.

A program might have a number of different ways of referencing a particular area. In this case, the program can issue multiple pairs of REFPAT INSTALL and REFPAT REMOVE macros for that area.

Each pattern, as defined on REFPAT INSTALL, is associated with the task that represents the caller. A task can have up to 100 reference patterns defined for multiple data areas at one time, but cannot have more than one pattern defined for the same area. Other tasks can specify a different reference pattern for the same data area. REFPAT REMOVE removes the association between the pattern and the task.

The data area can be in the primary address space or in a data space owned by a task that was dispatched in the primary address space. If the data area is in a data space, identify the data space through its STOKEN. You received the STOKEN either from DSPSERV or from another program.

Although REFPAT can be used for data structures other than arrays, for simplicity, examples in this chapter use REFPAT for an array or part of an array.

Reference pattern services for high-level language (HLL) and assembler language programs provide function similar to what REFPAT offers. For information about these services, see *Callable Services for High-Level Languages*.

How Does the System Handle the Data in an Array?

To evaluate the performance advantage REFPAT offers, you need to understand how the system handles a range of data that a program references. Consider the two-dimensional array in Figure 15-2 that is shown in row-major order and in order of increasing addresses. This array has 1024 columns and 1024 rows and each element is eight bytes in size. Each number in Figure 15-2 represents one element. The size of the array is 1048576 elements for a total of 8388608 bytes. For simplicity, assume the array is aligned on a page boundary. Assume, also, that the array is not already in central storage. The program references each element in the array in a forward direction (that is, in order of increasing addresses) starting with the first element in the array.

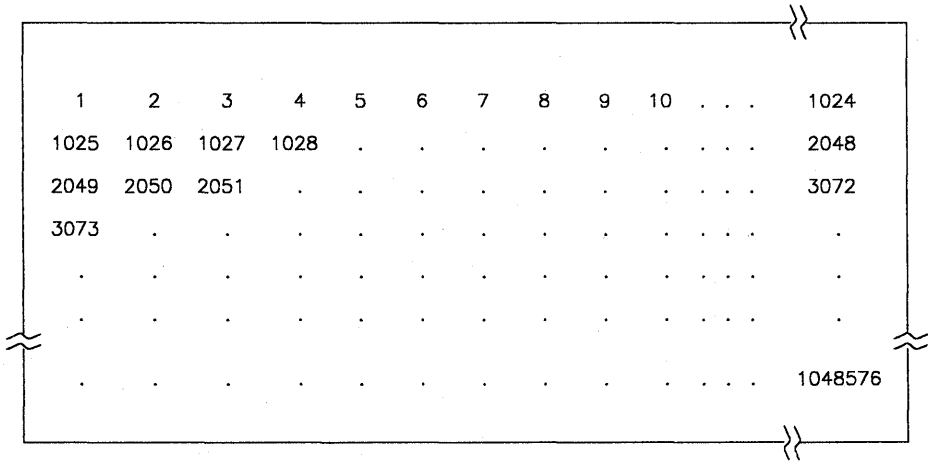
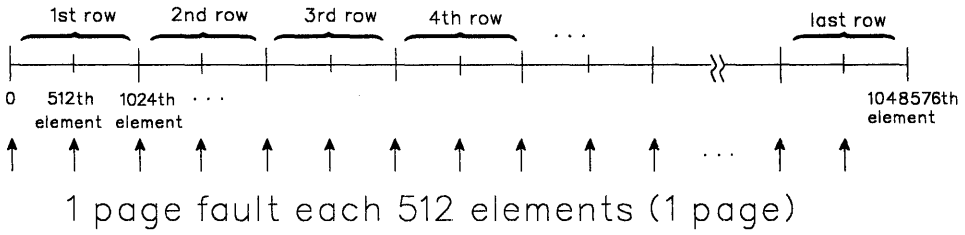


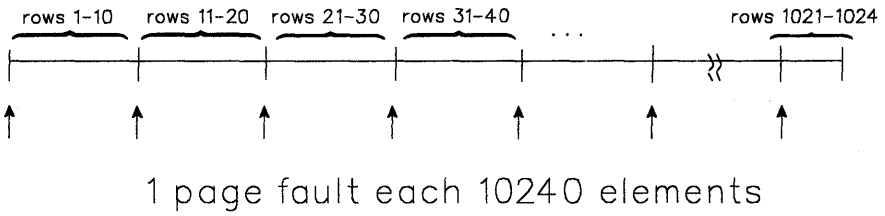
Figure 15-2. Example of using REFPAT with a Large Array

First, consider how the system brings data into central storage without using the information REFPAT provides. At the first reference of the array, the system takes a page fault and brings into central storage the page (of 4096 bytes) that contains the first element. After the program finishes processing the 512th element (4096÷8) in the array, the system takes another page fault and brings in a second page. To provide the data for this program, the system takes two page faults for each row. The following linear representation shows the elements in the array and the page faults the system takes as the program processes through the array.



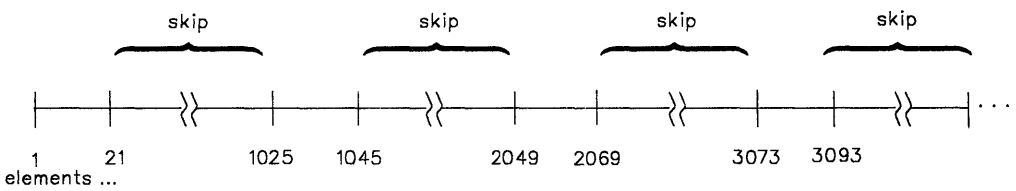
By bringing in one page at a time, the system takes 2048 page faults (8388608÷4096), each page fault adding to the elapsed time of the program.

Suppose, through REFPAT, the system knew in advance that a program would be using the array in a consistently forward direction. The system could then assume that the program's use of the pages of the array would be sequential. To decrease the number of page faults, each time the program requested data that was not in central storage, the system could bring in more than one page at a time. Suppose the system brought the next 20 consecutive pages (81920 bytes) of the array into central storage on each page fault. In this case, the system takes not 2048 page faults, but 103 (8388608÷81920=102.4). Page faults occur in the array as follows:



The system brings in successive pages only to the end of the array.

Consider another way of referencing this same array. The program references the first twenty elements in each row, then skips over the last 1004 elements, and so forth through the array. REFPAT allows you to tell the system to bring in only the pages that contain the data in the first 20 columns of the array, and not the pages that contain only data in columns 21 through 1024. In this case, the reference pattern includes a repeating gap of 8032 bytes (1004x8) every 8192 bytes (1024x8). The pattern looks like this:



The grouping of consecutive bytes that the program references is called a **reference unit**. The grouping of consecutive bytes that the program skips over is called a **gap**. Reference units and gaps alternate throughout the data area. The reference pattern is as follows:

- The reference unit is 20 elements in size — 160 consecutive bytes that the program references.
- The gap is 1004 elements in size — 8032 consecutive bytes that the program skips over.

Figure 15-3 illustrates this reference pattern and shows the pages that the system does not bring into central storage.

What Pages Does the System Bring in When a Gap Exists?

When no gap exists, the system brings into central storage all the pages that contain the data in the range you specify on REFPAT. When there is a gap, the answer depends on the size of the gap, the size of the reference unit, and the alignment of reference units and gaps on page boundaries. The following examples illustrate those factors.

Example 1: The following illustration shows the 1024-by-1024 array of eight-byte elements, where the program references the first 20 elements in each row and skips over the next 1004 elements. The reference pattern, therefore, includes a reference unit of 160 bytes and a gap of 8032 bytes. The reference units begin on every other page boundary.

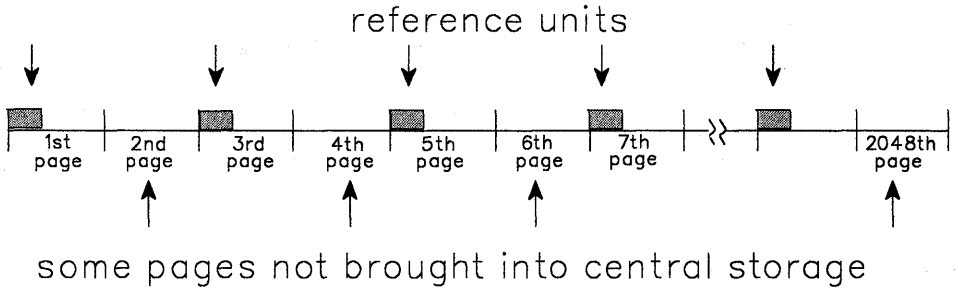
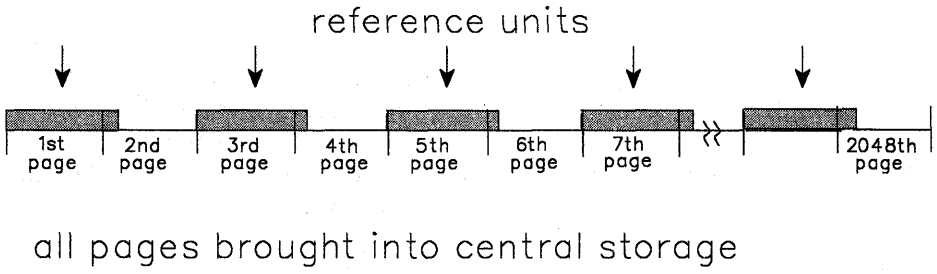


Figure 15-3. Illustration of a Reference Pattern with a Gap

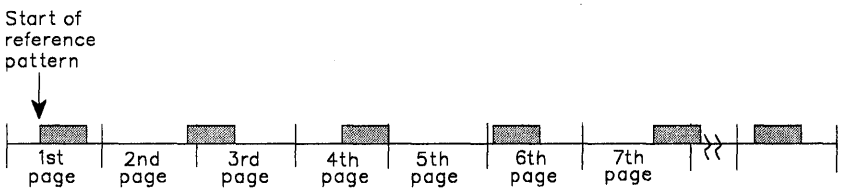
Every other page of the data does not come into central storage; those pages contain only the "skipped over" data.

Example 2: The reference pattern includes a reference unit of 4800 bytes and a gap of 3392 bytes. The example assumes that the area to be referenced starts on a page boundary.



Because each page contains data that the program references, the system brings in all pages.

Example 3: The area to be referenced does not begin on a page boundary. The reference pattern includes a reference unit of 2000 bytes and a gap of 5000 bytes. Because the reference pattern includes a gap, the first byte of the reference pattern must begin a reference unit, as the following illustration shows:



most pages brought into central storage

Because the gap is larger than 4095 bytes, some pages do not come into central storage. Notice that the system does not bring in the fifth page.

Summary of how the size of the gap affects the pages the system brings into central storage:

- If the gap is less than 4096 bytes, the system has to bring into central storage all pages of the array. (See Example 2.)
- If the gap is greater than 4095 bytes and less than 8192, the system might not have to bring in certain pages. Pages that contain only data in the gap are not brought in. (See Examples 1 and 3.)
- If the gap is greater than 8191 bytes, the system definitely does not have to bring in certain pages that contain the gap.

Using the REFPAT Macro

On the REFPAT macro, you tell the system:

- The starting and ending addresses of the data area to be referenced
- The reference pattern
- The number of reference units the system is to bring into central storage on a page fault.

Specify the reference pattern carefully on REFPAT. If you identify a pattern and do not adhere to it, the system will have to work harder than if you had not issued the macro. "Defining the Reference Pattern" on page 15-11 can help you define the reference pattern.

The system will not process the REFPAT macro unless the values you specify can result in a performance gain for your program. To make sure the system processes the macro, ask the system to bring in more than three pages (that is, 12288 bytes) on each page fault. "Choosing the Number of Bytes on a Page Fault" on page 15-12 can help you meet that requirement.

Identifying the Data Area and Direction of Reference

On the PSTART and PEND parameters, you specify the starting and ending addresses of the area to be referenced. If the reference is in a backward direction, the ending address will be smaller than the starting address.

PSTART identifies the first byte of the data area that the program references with the defined pattern; PEND identifies the last byte.

When a gap exists, define PSTART and PEND according to the following rules:

- If direction is forward, PSTART must be the first byte (low-address end) of a reference unit; PEND can be any part of a reference unit or a gap.
- If direction is backward, PSTART must be the last byte (high-address end) of a reference unit; PEND can be any part of a reference unit or a gap.

Figure 15-4 illustrates a reference pattern that includes a reference unit of 2000 bytes and a gap of 5000 bytes. When direction is forward, PSTART must be the beginning of a reference unit. PEND can be any part of a gap or reference unit.

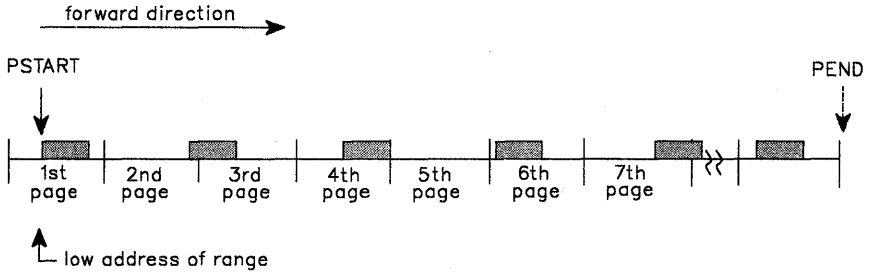


Figure 15-4. Illustration of Forward Direction in a Reference Pattern

Figure 15-5 illustrates the same reference pattern and the same area; however, the direction is backward. Therefore, PSTART must be the last byte of a reference unit and PEND can be any part of a gap or reference unit.

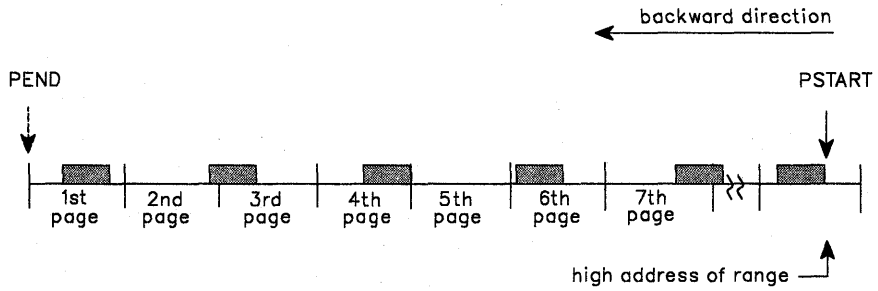


Figure 15-5. Illustration of Backward Direction in a Reference Pattern

If the data area is in a data space, use the STOKEN parameter to identify the data space. You received the STOKEN of the data space from another program or from the DSPSERV macro when you created the data space. STOKEN=0, the default, tells the system that the data is in the primary address space.

Defining the Reference Pattern

This section assumes that your program's reference pattern meets the basic requirement of consistent direction. Figure 15-6 identifies two reference patterns that characterize most of the reference patterns that REFPAT applies to. The marks on the line indicate referenced elements.

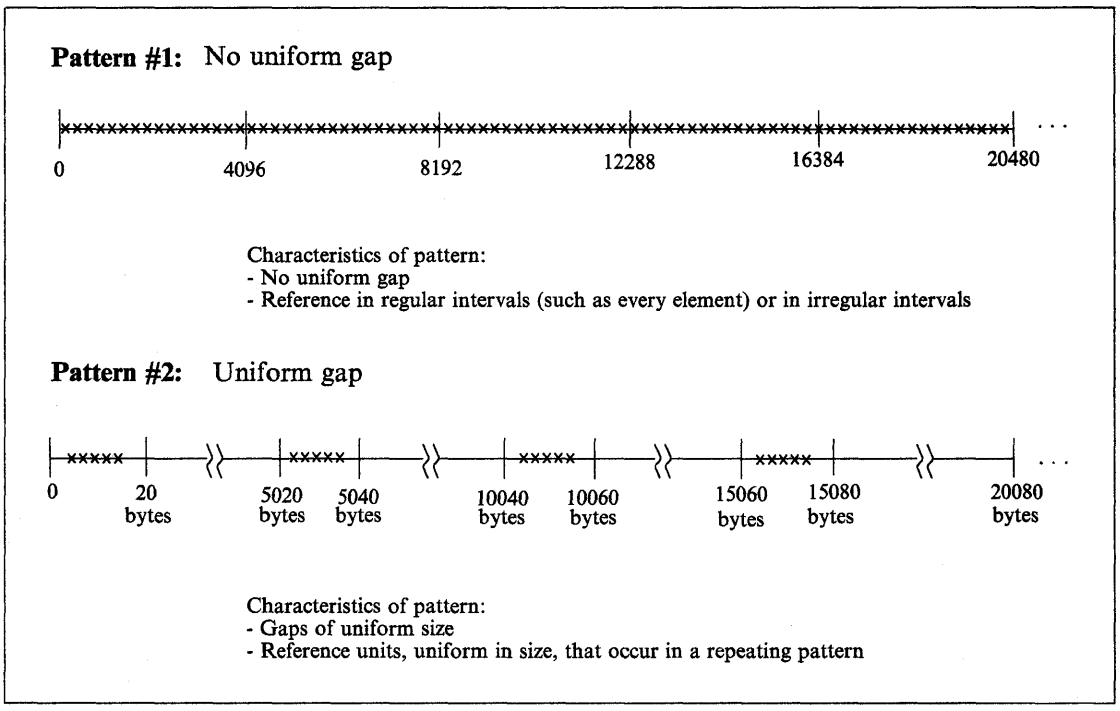


Figure 15-6. Two Typical Reference Patterns

How you define the reference pattern depends on whether your program's reference pattern is like Pattern #1 or Pattern #2.

- **With Pattern #1 where no uniform gap exists**, the program uses every element, every other element, or at least most elements on each page of array data. No definable gap exists. Do not use REFPAT if the reference pattern is irregular and includes skipping over many areas larger than a page.
  - The UNITSIZE parameter alone identifies the reference pattern. (Either omit the GAP parameter or take the default, GAP=0.) UNITSIZE indicates the number of bytes you want the system to use as a reference unit. Look at logical groupings of bytes, such as one row, a number of rows, or one element, if the elements are large in size. Or, you might choose to divide up the total area, bringing in all the data on a certain number of page faults.
  - The UNITS parameter tells the system how many reference units to try to bring in on a page fault. For a reference pattern that begins on a page boundary and has no gap, the total number of bytes the system tries to bring into central storage at a time is the value on UNITSIZE times the number on UNITS, rounded up to the nearest multiple of 4096. See "Choosing the Number of Bytes on a Page Fault" on page 15-12 for more information on how to choose the total number of bytes.

- **With Pattern #2 where a uniform gap exists**, you tell the system the sizes of reference units and gaps.
  - UNITSIZE and GAP parameters identify the reference pattern. Pattern #2 in Figure 15-6 on page 15-11 includes a reference unit of 20 bytes and a gap of 5000 bytes. Because the gap is greater than 4095, some pages of the array might not come into central storage.
  - The UNITS parameter tells the system how many reference units to try to bring into central storage at a time. "What Pages Does the System Bring in When a Gap Exists?" on page 15-8 can help you understand how many bytes come into central storage at one time.

Although the system brings in pages 4096 bytes at a time, you do not have to specify GAP, UNITS, and UNITSIZE values in increments of 4096.

Choosing the Number of Bytes on a Page Fault

An important consideration in using REFPAT is how many bytes to ask the system to bring in on a page fault. To determine this, you need to understand some factors that affect the performance of your program.

Pages do not stay in central storage if they are not referenced frequently enough and other programs need that central storage. The longer it takes for a program to begin referencing a page in central storage, the greater the chance that the page has been moved out to auxiliary storage before being referenced. When you tell the system how many bytes it should try to bring into central at one time, you have to consider the following:

1. **Contention for central storage**
   Your program contends for central storage along with all other submitted jobs. The greater the size of central storage, the more bytes you can ask the system to bring in on a page fault. The system responds to REFPAT with as much of the data you request as possible, given the availability of central storage.
2. **Contention for processor time**
   Your program contends for the processor's attention along with all other submitted jobs. The more competition, the less the processor can do for your program and the smaller the number of bytes you should request.
3. **The elapsed time of processing one page of your data**
   How long it takes a program to process a page depends on the number of references per page and the elapsed time per reference. If your program uses only a small percentage of elements on a page and references them only once or twice, the program completes its use of pages quickly. If the processing of each referenced element includes processor-intensive operations or a time-intensive operation, such as I/O, the time the program takes to process a page gets longer.

Conditions might vary between the peak activity of the daytime period and the low activity of other periods. For example, you might be able to request a greater number in the middle of the night than during the day.

What if you specify too many bytes? What if you ask the system to bring in so many pages that, by the time your program needs to use some of those pages, they have left central storage? The answer is that the system will have to bring them in again. This action causes an extra page fault and extra system overhead and reduces the benefit of reference pattern services.

For example, suppose you ask the system to bring in 204800 bytes, or 50 pages, at a time. But, by the time your program begins referencing the data on the 30th page, the system has moved that page and the ones after it out of central storage. (It moved them out because the program did not use them soon enough.) In this case, your program has lost the benefit of moving the last 21 pages in. Your program would get more benefit by requesting fewer than 30 pages.

What if you specify too few bytes? If you specify too small a number, the system will take more page faults than it needs to and you are not taking full advantage of reference pattern services.

For example, suppose you ask the system to bring in 40960 bytes (10 pages) at a time. Your program's use of each page is not time-intensive, meaning that the program finishes using the pages quickly. The program can request a number greater than 10 without causing additional page faults.

IBM recommends that you use one of the following approaches, depending on whether you want to involve your system programmer in the decision.

- The first approach is the easier one. Choose a conservative number of bytes, around 81920 (20 pages), and run the program. Look for an improvement in the elapsed time. If you like the results, you might increase the number of bytes. If you continue to increase the number, at some point you will notice a diminishing improvement or even an increase in elapsed time. Do not ask for so much that your program or other programs suffer from degraded performance.
- A second approach is for the program that needs very significant performance improvements — those programs that require amounts in excess of 50 pages. If you have such a program, you and your system programmer must examine the program's elapsed time, paging speeds, and processor execution times. In fact, the system programmer can tune the system with your program in mind and provide needed paging resources. *Initialization and Tuning* can provide information on tuning the system.

  REFPAT affects movement of pages from auxiliary **and** expanded storage to central storage. To gain insight into the effectiveness of your reference patterns, you and your system programmer will need the kind of information that the SMF Type 30 record provides. A Type 30 record reports counts of pages moved (between expanded and central and between auxiliary and central) in anticipation of your program's use of those pages. It also provides elapsed time values. Use this information to calculate rates of movement in determining whether to specify a very large number of bytes — for example, an amount greater than 204800 bytes (50 pages).

Examples of Using REFPAT to Define a Reference Pattern

To clarify the relationships between the UNITSIZE, UNITS, and GAP parameters, this section contains three examples of defining a reference pattern. So that you can compare the three examples with what the system does without information from REFPAT, the following REFPAT invocation approximates the system's normal paging operation:

```
REFPAT  INSTALL,PSTART=. . .,PEND=. . .,UNITSIZE=4096,GAP=0,UNITS=1
```

Each time the system takes a page fault, it brings in 4096 bytes, the system's reference unit. It brings in one reference unit at a time.

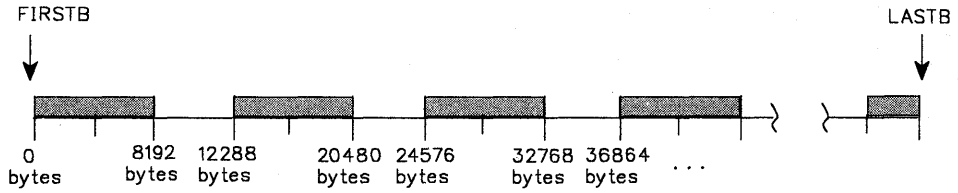
Example 1: The program processes an array in a consistently forward direction from one reference unit to the next. The processing of each element is fairly simple. The program runs during the peak hours and many programs compete for processor time and central storage. A reasonable value to choose for the number of bytes to be brought into central storage on a page fault might be 80000 bytes (around 20 pages). A logical grouping of bytes (the UNITSIZE parameter) is 4000 bytes. The following REFPAT macro communicates this pattern to the system:

```
REFPAT  INSTALL,PSTART=FIRSTB,PEND=LASTB,UNITSIZE=4000,GAP=0,UNITS=20
```

Example 2: The program performs the same process as in the first example, except the program references few elements on each page. The program runs during the night hours when contention for the processor and for central storage is light. In this case, a reasonable value to choose for the number of bytes to come into central storage on a page fault might be 200000 bytes (around 50 pages). UNITSIZE can be 4000 bytes and UNITS can be 500. The following REFPAT macro communicates this pattern:

```
REFPAT  INSTALL,PSTART=FIRSTB,PEND=LASTB,UNITSIZE=4000,GAP=0,UNITS=500
```

Example 3: The program references in a consistently forward direction through the same large array as in the second example. The pattern of reference includes a gap. The program references 8192 bytes, then skips the next 4096 bytes, references the next 8192 bytes, skips the next 4096 bytes, and so forth throughout the array. The program chooses to bring in data eight pages at a time. Because of the size of the gap and the placement of reference units and gaps on page boundaries, the system does not bring in the data in the gaps. The following illustration shows this reference pattern:



The following REFPAT macro reflects this reference pattern:

```
REFPAT  INSTALL,PSTART=FIRSTB,PEND=LASTB,UNITSIZE=8192,GAP=4096,UNITS=4
```

where the system is to bring into central storage 32768 (4x8192) bytes on a page fault.

Removing the Definition of the Reference Pattern

When a program is finished referencing the data area in the way you specified on the REFPAT INSTALL macro, use REFPAT REMOVE to tell the system to return to normal paging. On the PSTART and PEND parameters, you specify the same values that you specified on the PSTART and PEND parameters that defined the reference pattern for the area. If you used the STOKEN parameter on REFPAT INSTALL, use it on REFPAT REMOVE.

The following REFPAT invocation removes the reference pattern that was defined in Example 3 in "Examples of Using REFPAT to Define a Reference Pattern":

```
REFPAT  REMOVE,PSTART=FIRSTB,PEND=LASTB
```

Chapter 16. Timing and Communication

This chapter describes timing services and communication services. Use timing services to obtain the present date and time or for interval timing. Interval timing lets you set a time interval, test how much time is left in the interval, or cancel the interval. Use communication services to send messages to the system operator, to TSO terminals, and to the system log.

Obtaining Date and Time of Day

The operator is responsible for initially supplying the correct date and the time of day in terms of a 24-hour clock. You request the date and time of day using the TIME macro. The control program returns the date in register 1 and the time of day in register 0 or in a doubleword that you supply if you specify the MIC or STCK parameter.

All references to time of day use the time-of-day (TOD) clock, a 64-bit binary counter. The TOD clock runs continuously while the power is on, and the clock is not affected by the system-stop conditions. The operator normally sets the clock only after an interruption of CPU power has caused the clock to stop, and restoration of power has restarted it. The operator sets the clock during system initialization in response to a system message. (For more information about the TOD clock, see *Principles of Operation*.)

Interval Timing

Time intervals can be established for any task in the job step through the use of the STIMER or STIMERM SET macros. The time remaining in an interval established via the STIMER macro can be tested or cancelled through the use of TTIMER macro. The time remaining in an interval established via the STIMERM SET macro can be cancelled or tested through the use of the STIMERM CANCEL or STIMERM TEST macros.

The value of the CPU timer can be obtained by using the CPUTIMER macro. The CPU timer is used to track task-related time intervals.

The TASK, REAL, or WAIT parameters of the STIMER macro and the WAIT=YES|NO parameter of the STIMERM SET macro specify the manner in which the time interval is to be decreased. REAL and WAIT indicate the interval is to be decreased continuously, whether the associated task is active or not. TASK indicates the interval is to be decreased only when the associated task is active. STIMERM SET can establish real time intervals only.

If REAL or TASK is specified on STIMER or WAIT=NO is specified on STIMERM SET, the task continues to compete with the other ready tasks for control; if WAIT is specified on STIMER, or WAIT=YES is specified on STIMERM SET, the task is placed in a WAIT condition until the interval expires, at which time the task is placed in the ready condition.

When TASK or REAL is specified on STIMER or WAIT=NO is specified on STIMERM SET, the address of an asynchronous timer completion exit routine can also be specified. This routine is given control sometime after the time interval completes. The delay is dependent on the system's work load and the relative dispatching

priority of the associated task. If an exit routine is not specified, there is no notification of the completion of the time interval. The exit routine must be in virtual storage when specified, must save and restore registers as well as return control to the address in register 14.

Timing services does not serialize the use of asynchronous timer completion routines.

Figure 16-1 shows the use of a time interval when testing a new loop in a program. The STIMER macro sets a time interval of 5.12 seconds, which is to be decreased only when the task is active, and provides the address of a routine called FIXUP to be given control when the time interval expires. The loop is controlled by a BXLE instruction.

```
        .
        .
        STIMER TASK,FIXUP,BINTVL=TIME   Set time interval
LOOP    ...
        TM      TIMEXP,X'01'  Test if FIXUP routine entered
        BC      1,NG          Go out of loop if time interval expired
        BXLE    12,6,LOOP     If processing not complete, repeat loop
        TTIMER  CANCEL        If loop completes, cancel remaining time
        .
        .
NG      ...
        .
        .
        USING   FIXUP,15      Provide addressability
FIXUP   SAVE    (14,12)       Save registers
        OI      TIMEXP,X'01'  Time interval expired, set switch in loop
        .
        .
        RETURN  (14,12)       Restore registers
        .
        .
TIME    DC      X'00000200'   Timer is 5.12 seconds
TIMEXP  DC      X'00'         Timer switch
```

Figure 16-1. Interval Processing

The loop continues as long as the value in register 12 is less than or equal to the value in register 6. If the loop stops, the TTIMER macro causes any time remaining in the interval to be canceled; the exit routine is not given control. If, however, the loop is still in effect when the time interval expires, control is given to the exit routine FIXUP. The exit routine saves registers and turns on the switch tested in the loop. The FIXUP routine could also print out a message indicating that the loop did not go to completion. Registers are restored and control is returned to the control program. The control program returns control to the main program and execution continues. When the switch is tested this time, the branch is taken out of the loop. Caution should be used to prevent a timer exit routine from issuing an STIMER specifying the same exit routine. An infinite loop may occur.

The priorities of other tasks in the system may also affect the accuracy of the time interval measurement. If you code REAL or WAIT, the interval is decreased continuously and may expire when the task is not active. (This is certain to happen when WAIT is coded.) After the time interval expires, assuming the task is not in the

wait condition for any other reason, the task is placed in the ready condition and then competes for CPU time with the other tasks in the system that are also in the ready condition. The additional time required before the task becomes active will then depend on the relative dispatching priority of the task.

The STIMER macro should not be issued while a BTAM OPEN or LINE OPEN operation is in progress, since the BTAM OPEN LINE routines also use STIMER. STIMER should not be issued before invoking dynamic allocation because dynamic allocation can also issue STIMER.

Obtaining Accumulated Processor Time

The TIMEUSED macro enables you to record execution times and to measure performance. TIMEUSED returns the amount of processor or vector time a task (TCB) has used since being created (attached).

TIMEUSED is available to unlocked programs running without FRRs.

Example of measuring performance with TIMEUSED macro:

Use TIMEUSED to measure the efficiency of a routine or other piece of code. If you need to sort data, you may now code several different sorting algorithms, and then test each one. The logic for a test of one algorithm might look like this:

1. Issue TIMEUSED
2. Save old time
3. Run sort algorithm
4. Issue TIMEUSED
5. Save new time
6. Calculate time used (new time - old time)
7. Issue a WTO with the time used and the algorithm used.

After running this test scenario for all of the algorithms available, you can determine which algorithm has the best performance.

Note: The processor time provided by TIMEUSED does not include any activity for execution in SRB mode (such as I/O interrupt processing).

Writing and Deleting Messages (WTO, WTOR, DOM, and WTL)

The WTO and the WTOR macros allow you to write messages to the operator. The WTOR macro also allows you to request a reply from the operator. The DOM macro allows you to delete a message that is already written to the operator. Only standard, printable EBCDIC characters, shown in Figure 16-2, appear on the MCS console. All other characters are replaced by blanks. If the terminal does not have dual-case capability, it prints lowercase characters as uppercase characters.

Hex Code	EBCDIC Character	Hex Code	EBCDIC Character	Hex Code	EBCDIC Character	Hex Code	EBCDIC Character
40	(space)	7B	#	99	r	D5	N
4A	¢	7C	@	A2	s	D6	O
4B	.	7D	'	A3	t	D7	P
4C	<	7E	=	A4	u	D8	Q
4D	(	7F	"	A5	v	D9	R
4E	+	81	a	A6	w	E2	S
4F	|	82	b	A7	x	E3	T
50	&	83	c	A8	y	E4	U
5A	!	84	d	A9	z	E5	V
5B	$	85	e	C1	A	E6	W
5C	*	86	f	C2	B	E7	X
5D	)	87	g	C3	C	E8	Y
5E	;	88	h	C4	D	E9	Z
5F	¬	89	i	C5	E	F0	0
60	-	91	j	C6	F	F1	1
61	/	92	k	C7	G	F2	2
6B	,	93	l	C8	H	F3	3
6C	%	94	m	C9	I	F4	4
6D	—	95	n	D1	J	F5	5
6E	>	96	o	D2	K	F6	6
6F	?	97	p	D3	L	F7	7
7A	:	98	q	D4	M	F8	8

Figure 16-2. Characters Printed or Displayed on an MCS Console

Notes:

1. If the display device or printer is managed by JES3, the following characters are also translated to blanks:
   | ! ; ¬ : "
2. The system recognizes the following hexadecimal representations of the U.S. national characters: @ as X'7C'; $ as X'5B'; and # as X'7B'. In countries other than the U.S., the U.S. national characters represented on terminal keyboards might generate a different hexadecimal representation and cause an error. For example, in some countries the $ character generates a X'4A'.

There are two basic forms of the WTO macro: the single-line form, and the multiple-line form.

The following should be considered when issuing multiple-line WTO messages (MLWTO).

- By default, only the first line of a multiple-line WTO message is passed to the installation-written WTO exit routine. The user exit can request to see all subsequent lines of a multi-line message.
- When a console switch takes place, unended multiple-line WTO messages and multiple-line WTO messages in the process of being written to the original console are not moved to the new console.
- When a hardcopy switch takes place from the system log to an active operator's console, MLWTO messages in the process of being written to the system log are not moved to the new hard copy device.
- If register 0 contains a value other than 0, the value is a console ID, if you specified MCSFLAG=REG0 without specifying CONSNAME or CONSID.
- When the system hard copy log is an active operator's console, only the hard copy versions of multiple-line messages are written to the console.

See the macros section for an explanation of the parameters in the single-line and multiple-line forms of the WTO macro.

Routing the Message

The ROUTCDE parameter allows you to specify the routing code or codes for a WTO and WTOR message. The routing codes determine which console or consoles receive the message. Each code represents a predetermined subset of the consoles that are attached to the system, and that are capable of displaying the message. It is up to the user to define the consoles that belong to each routing code. WTO and WTOR allow routing codes from 1 to 128.

During system initialization, each operator's console in the system is assigned routing codes that correspond to the functions that the installation wants that console to perform. When any of the routing codes assigned to a message match any of the routing codes assigned to a console, the message is sent to that console.

Disposition of the message is indicated through the descriptor codes specified in the WTO macro. Descriptor codes classify WTO messages so that they can be properly presented on, and deleted from, display devices. Each WTO macro should contain at least one descriptor code. The descriptor code is not printed or displayed as part of the message text.

If the user supplies a descriptor code in the WTO macro, an indicator is inserted at the start of the message. The indicators are: a blank, an at sign (@), an asterisk (*), or a blank followed by a plus sign (+). The indicator inserted in the message depends on the descriptor code that the user supplies and whether the user is a privileged or APF-authorized program or a non-authorized problem program. Figure 16-3 shows the indicator that is used for each descriptor code.

Descriptor Code	Privileged or APF-Authorized Program	Non-Authorized Problem Program
1	*	@
2	*	@
3-10	blank	blank+
11	*	@
12-16	blank	blank+

Figure 16-3. Descriptor Code Indicators

The indicator @ or * informs operators that they must take some immediate or critical eventual action. A critical eventual action is an action that the operator must perform, as soon as possible, in response to a critical situation during the operation of the system. For example, if the dump data set is full, the operator is notified to mount a new tape on a specific unit. This is considered a critical action because no dumps can be taken until the tape is mounted; it is eventual rather than immediate because the system continues to run and processes jobs that do not require dumps.

Action messages to the operator, which are identified by the @ or * indicator, can be individually suppressed by the installation. When a program invokes WTO or WTOR to send a message, the system determines if the message is to be suppressed. If the message is to be suppressed, the system writes the message to the hardcopy log and the operator does not receive it on the screen. For more information on suppressing messages, see *Planning: Operations*.

If a problem program issues a message with descriptor code of 1 or 2, the message is deleted at address space or task termination. For more information concerning routing and descriptor codes, see *Routing and Descriptor Codes*.

If an application that uses WTO needs to alter a message each time the message is issued, the list form of the WTO macro may be useful. You can alter the message area, which is referenced by the WTO parameter list, before you issue the WTO. The message length, which appears in the WTO parameter list, does not need to be altered if you pad out the message area with blanks.

A sample WTO macro is shown in Figure 16-4.

```
Single-line WTO  'BREAKOFF POINT REACHED.  TRACKING COMPLETED.',
format           ROUTCDE=14,DESC=7

Multiple-   WTO   ('SUBROUTINES CALLED',C),
line format       ('ROUTINE TIMES CALLED',L),('SUBQUER',D),
(list form)       ('ENQUER',D),('WRITER',D),
                  ('DQUER',DE),
                  ROUTCDE=(2,14),DESC=(7,8),MF=L
```

Figure 16-4. Writing to the Operator

The MCSFLAG=BUSYEXIT parameter determines what happens if no message buffers are available. If you specify BUSYEXIT and no console buffers for either MCS or JES3 are available, or if you specify BUSYEXIT and there is a JES3 staging area excess, the WTO is terminated. Control is returned to the issuer with a return code of X'20' and a reason code in register 0. The reason code is equal to the number of active WTO buffers for the issuer's address space. If you do not specify BUSYEXIT, WTO processing may place the WTO invocation in a wait state until WTO buffers are again available.

Another alternative for routing a message is to use the CONSID or CONSNAME parameter. These mutually exclusive parameters let you specify a field or register that contains the four-byte ID or pointer to an eight-byte console name of the console that is to receive the message. This is the preferred alternative to the MCSFLAG option of placing the console ID in register zero. When you issue a WTO or WTOR macro that uses both the CONSID or CONSNAME parameters and the ROUTCDE parameters, the message or messages will go to all of the consoles specified by both parameters.

Note: By using the various parameters of WTO or WTOR, you can route messages by routing code and console ID. See the description of the WTO or WTOR macro in *Assembler Programming Reference* for additional information.

Altering Message Text

If an application that uses WTO needs to alter the same message or numerous messages repetitively, using the TEXT parameter on the WTO macro may be useful. You can alter the message or messages in one of two ways:

- If you issue 3 different messages, all with identical parameters other than TEXT, you can create a list form of the macro, move the text into the list form, then execute the macro. Using the TEXT parameter you can use the standard form of the macro, and specify the address of the message text. By reducing the number of list and execute forms of the WTO macro in your code, you reduce the storage requirements for your program.

- If you need to modify a parameter in message text, using the TEXT parameter enables you to modify the parameter in the storage that you define in your program to contain the message text, rather than modify the WTO parameter list.

Using the TEXT parameter on WTO can reduce your program's storage requirements because of fewer lines of code or fewer list forms of the WTO macro.

To use the WTOR macro, code the message exactly as designated in the single-line WTO macro. (The WTOR macro cannot be used to pass multiple-line messages.) When the message is written, the control program adds a message identifier before the message to associate the reply with the message. The control program also inserts an indicator as the first character of all WTOR messages, thereby informing the operator that immediate action is required. You must, however, indicate the response desired. In addition, you must supply the address of the area in which the control program is to place the reply, and you must indicate the maximum length of the expected reply. The length of the reply may not be zero. You also supply the address of an event control block which the control program posts after the reply has been placed, left-adjusted, in your designated area.

You can also supply a command and response token, or CART, with any message. You may have received a CART as input in cases where you issued a message in response to a command. In these cases, you should specify this CART on any messages you issue. Using the CART guarantees that these messages are associated with the command.

A sample WTOR macro is shown in Figure 16-5. The reply is not necessarily available at the address you specified until the specified ECB has been posted.

```
         .
         .
         XC     ECBAD,ECBAD          Clear ECB
         WTOR   'STANDARD OPERATING CONDITIONS?  REPLY YES OR NO',
                REPLY,3,ECBAD,ROUTCDE=(1,15)
         WAIT   ECB=ECBAD
         .
         .
ECBAD    DC     F'0'                 Event control block
REPLY    DC     C'bbb'               Answer area
```

Figure 16-5. Writing to the Operator With a Reply

When a WTOR macro is issued, any console receiving the message has the authority to reply. The first reply received by the control program is returned to the issuer of the WTOR, providing the syntax of the reply is correct. If the syntax of the reply is not correct, another reply is accepted. The WTOR is satisfied when the control program moves the reply into the issuer's reply area and posts the event control block. Each console that received the original WTOR will also receive the accepted reply unless it's a security message. No console receives the accepted reply to a security message. A console with master authority may answer any WTOR, even if it did not receive the original message.

Writing a Multiple-Line Message

Issue messages consisting of multiple lines by using the WTO multiple-line capability to assure that all lines of a multiple-line message appear together and are not broken up by other single-line messages.

Embedding Label Lines in a Multiple-Line Message

Label lines provide column headings in tabular displays. You can change the column headings used to describe different sections of a tabular display by embedding label lines in the existing multiple-line WTO message for a tabular display.

Note: You cannot use the WTO macro to embed label lines. The WTO macro handles label lines at the beginning of the message only.

Communicating in a Sysplex Environment

The WTO macro allows applications to send messages to consoles within a sysplex, without having to be aware that more than one system is up and running. Each system is aware of all the consoles in the sysplex, of their system attachment, and of their routing codes. This data includes the status of the console (active or inactive), the system where it is currently active, and the console's name, ID, routing codes, message level, and message type.

You can direct a WTO to a specific console by specifying the console ID or name when the issuing the message. For example, you can use the CONSID, CONSNAME, or MCSFLAG=REG0 parameter on the WTO macro to direct the WTO to consoles defined by those parameters. If the console is not active anywhere within the sysplex, WTO will write the message to the system log unless it is an important information message, an action message or WTOR. Action messages, messages with descriptor code 12, and WTORs are written to the system log and then directed to consoles having the UD (undelivered) attribute for display.

You can also broadcast WTOs to all active consoles using MCSFLAG=BRDCST on the WTO macro. Unsolicited messages are directed by routing code, message level, and message type to the appropriate consoles anywhere within the sysplex. There may be some unsolicited messages that will not be queued to any console at a receiving system. In this case, all of the messages are written to the system log, and action messages are sent to the consoles with the UD attribute.

Writing to the Programmer

The WTO and the WTOR macros allow you to write messages to a programmer who is logged onto a TSO terminal, as well as to the operator. However, only the operator can reply to a WTOR message.

To write a message to the programmer, you must specify ROUTCDE=11 in the WTO or the WTOR macro.

Writing to the System Log

The system log consists of one SYSOUT data set on which the communication between the operator and the system is recorded. You can use the system log by coding the information that you wish to log in the "text" parameter of the WTL macro.

When the WTL macro is executed, the control program places your text in one of the buffers and, when the buffer is full, writes the buffer onto the system log data set. The control program writes the text of your WTL macro on the master console instead of on the system log if the system log is not active.

Although when using the WTL macro you code the message within apostrophes, the written message does not contain the apostrophes. The message can include any character that is valid for the WTO macro and is assembled and written the same way as the WTO macro. MCS routing codes and descriptor codes are not assigned, since they are not needed by the WTL macro.

Note: The exact format of the output of the WTL macro varies depending on the job entry system (JES2 or JES3) that is being used, the output class that is assigned to the log at system initialization, and whether DLOG is in effect for JES3. In JES3, system log entries are preceded by a prefix that includes a time stamp and routing information. If the combined prefix and message exceeds 126 characters, the log entry is split at the first blank or comma encountered when scanning backward from the 126th character of the combined prefix and message. See *JES3 Commands* for information about the format of the log entry when using JES3.

The WTO macro with the MCSFLAG=HRDCPY parameter also writes messages to the system log. Because WTO supplies more data than WTL, IBM recommends that you use it instead of WTL.

Deleting Messages Already Written

The DOM macro deletes the messages that were created using the WTO or WTOR macros. Depending on the timing of a DOM macro relative to the WTO or WTOR, the message may or may not have already appeared on the operator's console.

- When a message already exists on the operator screen, it has a format that indicates to the operator whether the message still requires that some action be taken. When the operator responds to a message, the message format changes to remind the operator that a response was already given. When DOM deletes a message, it does not actually erase the message. It only changes its format, displaying it like a non-action message.
- If the message is not yet on the screen, DOM deletes the message before it appears. The DOM processing does not affect the logging action. That is, if the message is supposed to be logged, it will be, regardless of when or if a DOM is issued. The message is logged in the format of a message that is waiting for operator action.

The program that generates an action message is responsible for deleting that message.

Note: Specifying the REPLY= parameter of the DOM macro causes an MNOTE warning message to be issued at assembly time. The MNOTE warns you that you are coding the REPLY= parameter, which is a function no longer supported in the system. If you code the REPLY= parameter and receive the MNOTE warning, remove the REPLY= parameter from your program and reassemble it. Programs

containing the REPLY= parameter that are already assembled do not need to be reassembled.

Identifying Messages to be Deleted

To identify the message or messages that you want to delete, you normally use the MSG, MSGLIST, or TOKEN parameters. When you issue a WTO or WTOR macro to write a given message to the operator, the system generates a message ID, which it returns in general register 1. To delete the message, you can issue the DOM macro with a MSG or MSGLIST parameter specifying the same system-generated message ID that WTO or WTOR returned in general register 1. If you specify MSGLIST (message list), then several message IDs can be associated with the delete request. The number of message IDs in the message list is defined by the COUNT parameter or it is defined by an 1 in the high order bit position of the last message ID in the list. The COUNT parameter cannot exceed 60.

On the other hand, the TOKEN parameter allows the message ID to be generated by the user rather than the system. When you issue WTO or WTOR with a TOKEN parameter, the system associates your TOKEN parameter with the message or all the messages that are written by this particular WTO or WTOR. Then you can issue DOM with the same TOKEN parameter to delete the message or all the messages associated with the token.

Retrieving Console Information (CONVCON macro)

Applications that either process commands or issue messages might need information about MCS or extended MCS consoles. CONVCON obtains information about these consoles.

Use the CONVCON macro to:

- Determine the name of a console when you supply the ID.
- Determine the ID of a console when you specify the name.
- Validate a console name or console ID.
- Validate a console area ID.
- Check if a console is active.

You must set up a parameter list, called CONV, before invoking the CONVCON macro. Depending upon the information you want, you must initialize certain fields in the CONVCON parameter list. CONVCON returns information in other fields of the parameter list. See *Diagnosis: Data Areas* for more information on the CONV parameter list, which is mapped by IEZVG200.

The following topics describe possible uses for the CONVCON tasks, and tell you how to fill in the parameter list for each task.

Determining the Name or ID of a Console

Installation operators and programmers previously referred to MVS consoles only by console IDs, which are defined in the CONSOLxx member of SYS1.PARMLIB. IBM strongly recommends that you use names when referring to MCS consoles. Using names can help operators and programmers:

- Remember which console they want to reference in commands or programs. For example, if your installation establishes one console to receive information about tapes, and uses the console name TAPE, operators and programmers can more easily remember TAPE than a console ID such as 03.

- Connect information in messages and the hardcopy log to the correct console. If your installation uses console IDs, operators and programmers might have difficulty identifying the console to which messages and hardcopy log information applies, because the system uses console names in messages and the hardcopy log.

Using console names rather than IDs can also avoid confusion when a console ID might change. If your installation has set up a sysplex environment, and uses console IDs to identify consoles, those IDs can change from one IPL to the next, or when systems are added or removed from the sysplex. A console ID is not guaranteed to be associated with one console for the life of a sysplex.

To determine the name of a console when you supply the ID, do the following:

1. Clear the CONVCON parameter list by setting it to zeros.
2. Initialize the following fields in the parameter list:
   - The version ID (CONVVRSN)
   - The acronym (CONVACRO)
   - The console ID (CONVID)
   - The flag indicating that you are supplying the console ID (flag CONVPID in CONVFLGS)
3. Issue the CONVCON macro.

When CONVCON completes, the console name is in the parameter list field CONVNAME, and register 15 contains a return code.

To determine the ID of a console when you supply the name, do the following:

1. Clear the CONVCON parameter list by setting it to zeros.
2. Initialize the following fields in the parameter list:
   - The version ID (CONVVRSN)
   - The acronym (CONVACRO)
   - The console name (CONVFLD)
   - The flag bit indicating that you are supplying the console name (flag CONVPFLD in CONVFLGS)

   The installation defines console names at initialization time in the CONSOLxx member of SYS1.PARMLIB. You can use the DISPLAY command to receive a list of defined names.
3. Issue the CONVCON macro.

When CONVCON completes, the console ID will be in the parameter field CONVID.

Validating a Console Name or ID and Checking if a Console Is Active

Before issuing a message to a specific console, you may want to determine whether that console exists by using CONVCON to validate the console name or ID.

You can use CONVCON to check if the console to which you are sending a command response is active. An application processing a command could use CONVCON to perform this check.

To check if a console is active, or to validate a console name or ID, do the following:

1. Clear the CONVCON parameter list by setting it to zeros.

2. Initialize the following fields in the parameter list:
   - The version ID (CONVVRSN)
   - The acronym (CONVACRO)
   - Either the console name (CONVFLD) or the console ID (CONVID) depending on what information you currently have. The installation defines console names at initialization time in the CONSOLxx member of SYS1.PARMLIB. You can use the DISPLAY command to receive a list of defined names.
   - The appropriate flag in CONVFLGS indicating whether you are specifying the console name (CONVPFLD) or the ID (CONVPID) as input.
3. Issue the CONVCON macro.

When CONVCON completes, register 15 contains a return code. If you receive the following return codes, check the reason code in register 0 for an explanation.

- If the return code is 0, the console name or ID is valid and the console is active.
- If the return code is 4, the name or ID is valid, but the console is not active.

If the return code is 8, the console name or ID is incorrect. Check the reason code in parameter list field CONVRSN for additional information.

Validating a Console Area ID

An area ID defines an out-of-line display area. An out-of-line display area is a predefined number of lines on a screen to which you can direct command responses, such as a response to a DISPLAY command. The area is for static displays, rather than in-line displays that roll on the screen.

If you want to issue a multi-line WTO to a specific out-of-line area on a console, and you want to know if the console and that area are available, you can use CONVCON to validate the console area ID. CONVCON validates that this area is available for use and that it does not already have a message in it.

To validate a console area ID, do the following:

1. Clear the CONVCON parameter list by setting it to zeros.
2. Initialize the following fields:
   - The version ID (CONVVRSN)
   - The CONVCON acronym (CONVACRO)
   - One of the following:
     - the console name with the area ID (CONVFLD-a)
     - the console ID and the area ID (CONVID and CONVAREA)
     - the console name and the area ID (CONVFLD and CONVAREA)

     The area ID can be one alphabetic character from A through J or Z.
   - The appropriate flag (CONVPID or CONVPFLD in CONVFLGS)
3. Issue the CONVCON macro.

When CONVCON completes, register 15 contains a return code. If the return code is 0 or 4, the reason code in the CONVRSN field of the parameter list indicates the validity of the area ID.

Coding Example

The following example will help you understand how to use CONVCON.

An operator or application has previously started a program by using the START command. A program may need to send a message to the console which issued the START command. However, that console may not currently be online or available.

The application can use the parameters in the input buffer control block (CIB) to determine the status of the console.

The application retrieves the console ID associated with the console which entered the START command. It retrieves the console ID from field CIBXCNID in the extension of the command input buffer control block, IEZCIB. CONVCON does the following:

- Determines whether the console is currently active
- Retrieves the name of the console associated with the ID
- Places the console name in an error message if the console is not currently active.

This example assumes:

- You have obtained storage for the CONV parameter list.
- You have mapped and referenced the command input buffer control block and its extension, to obtain the 4-byte console ID for input.
- You have placed the ID of the console to which CONVCON will send an error message in register 2.

```
SAMPLE   CSECT
WORK4    EQU   4
R2       EQU   2
REG12    EQU   12
REG15    EQU   15
         LR    REG12,REG15
         BALR  REG12,0
         USING *,REG12                    MODULE ADDRESSABILITY
CALLCONV EQU   *
         LA    WORK4,CONV                 ADDRESSABILITY - CONV
         USING CONV,WORK4
         XC    0(CONVPLEN,WORK4),0(WORK4) CLEAR PARAMETER LIST
         MVC   CONVACRO,ACNMCONV          ACRONYM - CONV
         MVI   CONVVRSN,CONVRID           CURRENT VERSION LEVEL
         OI    CONVFLGS,CONVPID           SET CONSOLE ID FLAG
         MVC   CONVID,CIBXCNID            SET 4 BYTE CONSOLE ID
         CONVCON (WORK4)                  ISSUE CONVCON
         C     REG15,EIGHT                RETURN CODE > 8?
         BH    ERROR                      YES, ERROR DURING CONVCON
         B     CONTABLE(REG15)            TAKE ACTION BY RETURN CODE
CONTABLE EQU   *                          ACTION TABLE
         B     CONOK                      CONSOLE ACTIVE. CONTINUE
         B     ISSUWTO                    INACTIVE. ISSUE WTO
         B     ERROR8                     UNDEFINED CONSOLE OR
*                                         SYNTAX ERROR. BRANCH
*                                         ELSEWHERE TO HANDLE
ISSUWTO  EQU   *
         MVC   UMSGCON,CONVNAME           MOVE CONSOLE NAME INTO
*                                         FIELD IN TEXT
         WTO   TEXT=UMSG1,MCSFLAG=RESP,CONSID=(R2),DESC=5
         B     EXIT                       EXIT ROUTINE
ERROR8   EQU   *                          BAD ID ON INPUT
CONOK    EQU   *                          SUBSEQUENT FUNCTION
EXIT     EQU   *                          EXIT LABEL
*-------------------------------------------------------------------*
*        CONSTANTS USED                                             *
*-------------------------------------------------------------------*
EIGHT    DC    F'8'
*-------------------------------------------------------------------*
*        MESSAGE TEXT FOR INACTIVE CONSOLE MESSAGE                  *
*-------------------------------------------------------------------*
UMSG1    EQU   *
UMSGLENG DC    XL2'56'                    LENGTH OF UMSG1
UMSG     DC    CL9'UMSG001I '             ERROR MESSAGE ID
UMSGTXT1 DC    C'REQUESTED CONSOLE '      TEXT SEGMENT 1
UMSGCON  DS    CL8' '                     CONSOLE NAME
UMSGTXT2 DC    C' NOT CURRENTLY ACTIVE'   TEXT SEGMENT 2
UMSGTXTE EQU   *                          END OF MESSAGE TEXT
UMSGTXTL EQU   UMSGTXTE-UMSG              MESSAGE TEXT LENGTH
ACNMCONV DC    C'CONV'                    ACRONYN FOR CONVCON PARMLIST
AREA     DC    C'A'                       OUT-OF-LINE AREA "A"
ZERO     EQU   0                          NUMERIC ZERO
         IEZVG200                         CONVCON PARMLIST
         IEZCIB                           CMDX PARMLIST
         END
```

Chapter 17. Translating Messages

The MVS message service (MMS) enables you to translate U.S. English messages into foreign languages. In addition, MVS-based applications can invoke MMS for one of two purposes:

- Rather than putting message text within application code or within a message CSECT, an application can have message text reside in the MMS run-time message files. The application can retrieve that message text by issuing the TRANMSG macro.
- An application can obtain and display foreign language message text by issuing the TRANMSG macro.

If you are routing system messages to a TSO/E extended MCS console, TSO/E will display translated messages in the primary language associated with the TSO/E session. If MMS is active, users of extended MCS consoles on TSO/E can select available languages for message translation and the system can display translated messages on the user's screen. TSO/E terminal users can also receive on their terminals translated TSO/E messages and the translated messages of any application that directs its messages to TSO/E and uses MMS services directly or through TSO/E services. To receive translated messages on TSO/E terminals, you must have TSO/E Version 2.2 installed on your system. Applications running on TSO/E can translate their messages through new PUTLINE support. For more information on PUTLINE see *TSO/E Version 2 Programming Services*.

The MVS message service can handle multi-line and multiple format messages. Multi-line messages are messages displayed over a number of lines on an output device. Multiple format messages are messages that have the same message ID, but have differing text content depending on the circumstances under which they are issued.

To prepare an application's messages for translation, perform the following tasks:

1. Create a partitioned data set (PDS) for the English version, and a PDS for the translated version of the application's messages. These data sets are the application's install message files.
2. Validate the application's install message files by running each PDS through the message compiler.
3. After a clean compile, incorporate your data sets into the system's install message files.
4. Update the system run-time message files by running the system's install message files through the message compiler.
5. Modify the application to exploit the translation service that MMS provides:
   - Use QRYLANG to determine which languages are available at your installation.
   - Use TRANMSG to obtain from MMS the translated version of an application's message or messages.

The installation can translate application messages into more than one language. See "Support for Additional Languages" on page 17-12.

Allocating Data Sets for an Application

For an application whose messages will be translated, you must allocate a partitioned data set (PDS) for each language in which the messages might appear. For example, if you want the messages to be available in both English and Japanese, you must allocate two data sets: one to contain the English message skeletons, and one to contain the Japanese.

See *JCL User's Guide* and *JCL Reference* for information about allocating data sets.

Creating Install Message Files

Each install message file must contain a version record and one or more message skeletons, and may contain any number of comment records throughout. The message compiler treats any record with the characters ".*" in columns 1 and 2 as a comment line and ignores it.

Creating a Version Record

The version record must be the first non-comment record in each install message file, and have the format shown in Figure 17-1.

Figure 17-1. Format of Version Record Fields

Columns	Contents and Description
1 & 2	".V" Identifies this record as a version record.
3-5	Three-character language code of the messages.
6-10	Reserved
11	Character field containing a Y or N, indicating whether this language contains double-byte characters (DBCS).
12-19	FMID (field maintenance identifier) applicable to the messages within the member, padded on the right with blanks.
20-27	Product identifier applicable to the messages within the member, padded on the right with blanks.
28-43	Service level applicable to the member, padded on the right with blanks.

The following is an example of a version record.

```
VENU00037NJBB44N1 5685-067005
```

Figure 17-2 illustrates the fields of the version record in the previous example.

Figure 17-2. Version Record Example

Columns	Example	Description
1 & 2	.V	Version record
3-5	ENU	Three-character language code
6-10		Reserved
11	N	DBCS indicator
12-19	JBB44N1b	FMID
20-27	5685-067	Product identifier
28-43	005	Service level information

Creating Message Skeletons

The rest of each install message file consists of message skeletons.

Each message requires one or more message skeletons. A message skeleton consists of a message key and message text, which can include substitution tokens. A message key consists of a message identifier, format number, and line number. A substitution token is a "place marker," identifying substitution data to MMS. MMS does not translate substitution tokens in the target language skeleton, but rather replaces them with actual substitution data.

Both &DSN1 and &FILE in the following examples are substitution tokens.

```
IACT0012W    001    DATASET &DSN1. NOT FOUND
IACT0012W    002    COULD NOT FIND DATASET &FILE.
```

Note: If the message skeleton you are creating contains a TIME, DATE, or DAY substitution token, the format must be defined in the system configuration member, CNLcccxx, for the language. See *MVS Initialization and Tuning Reference* for more information on these substitution tokens.

Message Skeleton Format

Each message skeleton must follow the column-oriented format shown in Figure 17-3.

Figure 17-3 (Page 1 of 2). Message Skeleton Fields

Columns	Contents and Description
1-10	Message identifier (*msgid*). A message identifier can be 1 to 10 characters long, padded with EBCDIC blanks, if necessary, so that it totals ten characters. The first character must be alphabetic. A message identifier cannot contain double-byte characters and cannot contain embedded blanks. Ensure that the message identifiers for your application program messages do not conflict with existing MVS message identifiers. To avoid conflict, do not begin a message identifier with the letters A through I. See *System Messages, Volume 1* for more information on MVS message identifiers. Examples of message identifiers are: • IKJ52301I • IEF12345W • HASP000 • IEF123I • OLDMSGID Note that MMS will remove the first character of any message identifier in the form x*msgid* before processing, and will replace it after processing. "x" is any character that is not uppercase alphabetic, such as $ or 1.

11 & 12	Line number (*ll*). If the message is a single-line message, leave columns 11 and 12 blank. For a multi-line message, assign line numbers sequentially within the message. Line numbers do not have to be contiguous. Valid numbers are 01 to 99.
	
	Ensure that message line numbers for a translated skeleton match the line numbers of the corresponding U.S. English message skeleton.
	
	Note: Ensure that corresponding skeletons (same message identifier and line number) of multi-line messages contain the same substitution tokens. For example, if substitution tokens *&1.* and *&2=date.* are on line 01 of a two-line U.S. English message skeleton, these tokens must appear on line 01 of a translated skeleton.
	
	You can translate multi-line messages that contain continuation lines, that is, only the first line contains a message identifier in the skeleton. The following is an example of a multi-line message:
	
	`MSGID01 THIS IS LINE ONE OF THIS MULTI-LINE MESSAGE`
	`THIS IS LINE TWO OF THIS MULTI-LINE MESSAGE`
	`THIS IS LINE THREE OF THIS MULTI-LINE MESSAGE`

Figure 17-3 (Page 2 of 2). Message Skeleton Fields

Columns	Contents and Description
13-15	Format number (*fff*). If only one format is defined for a particular message identifier, leave columns 13-15 blank. Use format numbers to maintain compatibility with existing messages. Using format numbers for new messages is not recommended. Format numbers distinguish among message skeletons that have one message identifier but have several different text strings or substitution tokens. The message identifier alone cannot identify the message as unique in these cases. The format number, together with the message identifier, identifies the message. If more than one format is defined for a particular message identifier, assign a unique format number to each skeleton for that message identifier. Valid numbers are 001 to 999. You do not have to assign the numbers sequentially. Ensure that the format number in the translated skeleton matches the format number in the U.S. English message skeleton. Each message ID might have several format numbers if that message has variable text.
16	Blank (b). Column 15 must contain an EBCDIC blank.

17 & 18	Translated line number (*mm*). If you are creating U.S. English skeletons, leave this field blank.
	If one line of a U.S. English message translates into more than one line of text in another language, you must provide additional lines for the translated version. Create one or more skeletons in the other language and assign a translated line number to each translated line. Valid translated line numbers are 01 to 99.
	Example:
	`IEFP0001  MAXIMUM PASSWORD ATTEMPTS BY SPECIAL USER &1. AT TERMINAL &2.`
	`IEFP0001  01 LE USER SPECIALE &1. A TERMINAL &2.`
	`02 ONT ENTRER PASSWORD TROP DE TEMPS`
	If a line of a U.S. English message translates to only one line, leave the translated line number blank.

19	Reserved.
20 +	Message text. See "Message Text in a Skeleton" on page 17-4

The following are examples of message skeletons.

```
msgid     llffftmm text

HASP001           ACCESS TO DATASET &DSN. DENIED

IACT0012W   001   DATASET &DSN1. NOT FOUND
IACT0012W   002   COULD NOT FIND DATASET &FILE.

HASP999I  01      ACCESS TO DATASET &1. DENIED:
HASP999I  02      USER INFORMED AT &2=DATE02. ON TERMINAL &3.

IEFA003F    001   USER &USERID. VIOLATED ACCESS RIGHTS TO DATASET &2. AT &3=TIME.
IEFA003F    002   &1=TIME.: USER &2. VIOLATED ACCESS RIGHTS TO DATASET &3.
```

Message Text in a Skeleton

Message text in a message skeleton must conform to certain format standards. The standards are as follows:

- Message text can be up to 256 bytes long including the message identifier, line number, and other fields.
- Message text can be upper-, lower-, or mixed case.
- Message text can be all single-byte character set (SBCS), all double-byte character set (DBCS), or a combination of both. Message text can contain substitution tokens.

Substitution tokens indicate substitution, or variable, data, such as a data set name or a date in the message text. Substitution tokens must start with a token start trigger character, an ampersand (&), and end with a token end trigger character, a period (.). These characters are part of the token and are not included in the message text display. You may include an ampersand (&) in the text as long as it does not have a period following it in the format of a substitution token. Substitution tokens must be SBCS characters and follow the form *&name*[*=format*] where:

name is the name of the substitution token. This name is an alphanumeric SBCS string. The name must not contain imbedded blanks or DBCS characters.

format is an optional format descriptor for the substitution token. Format descriptors are:

- TEXT for tokens other than dates and times (default format)
- DATExxxxxx for dates
- TIMExxxxxx for times
- DAY for the day of the week

If you use these format descriptors, you must also define them in the CNLcccxx parmlib member for the language. See *MVS Initialization and Tuning Reference* for more information on format descriptors.

The total length of *name* and *=format* must not be greater than 16 bytes.

If you do not include a format descriptor for a particular substitution token, the MVS message service treats the token as TEXT.

The date and time tokens are formatted according to the language. There are no defaults. You must supply your own formats in the CNLcccxx member.

Examples of substitution tokens are:

```
&1.
&USERID.
&1=DATE1.
&5=TIMESHORT.
```

Validating Message Skeletons

After creating message skeletons for both the U.S. English and translated version of each message, validate the skeletons. To validate the skeletons, run each of the application's install message files through the message compiler for syntax checking. Otherwise you might be adding incorrect skeletons to the files that MMS uses, and your messages might be either incorrectly translated or untranslatable. You should also validate skeletons when you add or change skeletons in an existing install message file.

To make sure your message skeletons are valid, complete the following process for each install message file:

1. Allocate storage for a run-time message file, which the compiler produces as output.
2. Compile the install message file by invoking the compiler.
3. Check the return code from the message compiler.

If the return code does not indicate a clean compile, use the compiler error messages to correct any errors in the skeletons. The compiler writes its error messages to the SYSPRINT data set. Then compile the install message file again.

The return code and error messages from the compiler are the only output you need to determine whether the message skeletons are correct. However, compiling an application's install message file also produces a formatted run-time message file. Before invoking the compiler, you must allocate storage for this run-time file, but you cannot use it as input for MMS. To make your application's messages available for translation, you must add your PDS to the system's install message files, and run those files through the compiler again.

Allocating Storage for a Validation Run-Time Message File

The data set you create for the run-time message file must be a linear VSAM data set that can be used as a data-in-virtual object. You must create one run-time file for each install message file for your application.

The amount of storage you will need to allocate for a validation run-time message file cannot be determined exactly. The amount of storage depends on the number of skeletons, the size of the skeletons, the number of substitution tokens within the skeletons, and the types of messages represented by the skeletons (single-line, multi-line, or multi-format). IBM recommends that, for a validation run-time message file, you allocate twice the amount of storage required for the install message file you are compiling. In most cases, this storage should be adequate.

To create the data set for the run-time message file, you need to specify the DEFINE CLUSTER function of IDCAMS with the LINEAR parameter. When you code the SHAREOPTIONS parameter for DEFINE CLUSTER, the cross-system value must be 4; that is, you may code SHAREOPTIONS as (1,3), (2,3), (3,3), or (4,3). Normally, use SHAREOPTIONS (1,3). For a complete explanation of SHAREOPTIONS, see the *Managing VSAM Data Sets*.

The following is a sample job that invokes access method services (IDCAMS) to create the linear data set named SYS1.ENURMF on the volume called MMSPK1. When IDCAMS creates the data set, it is empty. Note that there is no RECORDS parameter; linear data sets do not have records.

```
//DEFCLUS JOB  MSGLEVEL=(2,0),USER=IBMUSER
//*
//*    DEFINE DIV CLUSTER
//*
//DCLUST  EXEC PGM=IDCAMS,REGION=4096K
//SYSPRINT DD SYSOUT=*
//MMSPK1   DD UNIT=3380,VOL=SER=MMSPK1,DISP=OLD
//SYSIN    DD *
    DELETE (SYS1.ENURMF) CL PURGE
    DEFINE CLUSTER (NAME(SYS1.ENURMF) -
                    VOLUMES(MMSPK1) -
                    CYL(1 1) -
                    SHAREOPTIONS(1 3) -
                    LINEAR) -
           DATA     (NAME(SYS1.ENURMF.DATA))
/*
```

Figure 17-4. Sample job to invoke IDCAMS

Compiling Message Files

The message compiler creates a run-time message file from an install message file. You need to run the message compiler once for each language you install and each time you update the application's install message files. The compiler expects a PDS as input. If the compiler cannot process a message skeleton, it issues an error message. It also sets a return code. See "Checking the Message Compiler Return Codes" on page 17-9 for a description of compiler return codes.

Invoking the Message Compiler

The message compiler is an executable program. You can use a batch job, a TSO/E CLIST, or a REXX EXEC to invoke the message compiler. The syntax for each type of invocation follows. The meaning of the variables (shown in lowercase in the examples) follows the examples.

```
//COMPILE  EXEC  PGM=CNLCCPLR,
//               PARM=(lang,dbcs)
//SYSUT1   DD    DSN=msg_pds,DISP=SHR
//SYSUT2   DD    DSN=msg_div_obj,DISP=(OLD,KEEP,KEEP)
//SYSPRINT DD    SYSOUT=*
```

Figure 17-5. Using JCL to Invoke the Compiler

```
PROC 0
FREE DD(SYSUT1,SYSUT2,SYSPRINT)              /* FREE DD'S           */
ALLOC DD(SYSUT1) DSN('msg_pds') SHR          /* ALLOC INPUT FILE    */
ALLOC DD(SYSUT2) DSN('msg_div_obj') OLD      /* ALLOC OUTPUT FILE   */
ALLOC DD(SYSPRINT) DSN(*)                    /* ALLOC SYSPRINT      */
CALL 'SYS1.LINKLIB(CNLCCPLR)' ('lang,dbcs')
                                          /* CALL MESSAGE COMPILER */
SET &RCODE = &LASTCC                         /* SET RETURN CODE     */
FREE DD(SYSUT1,SYSUT2,SYSPRINT)              /* FREE FILES          */
EXIT CODE(&RCODE)                            /* EXIT                */
```

Figure 17-6. Using CLIST to Invoke the Compiler

```
/* MESSAGE COMPILER INVOCATION EXEC */

MSGCMPLR:

"FREE DD(SYSUT1,SYSUT2,SYSPRINT)"

"ALLOC DD(SYSUT1) DSN('"msg_pds"') SHR"
"ALLOC DD(SYSUT2) DSN('"msg_div_obj"') OLD"
"ALLOC DD(SYSPRINT) DSN(*)"

"CALL 'SYS1.LINKLIB(CNLCCPLR)' ('"lang,dbcs"')"

compiler_rc=rc

"FREE DD(MSGIN,MSGOUT,SYSPRINT)"

return(compiler_rc)
```

Figure 17-7. Using REXX to Invoke the Compiler

The lowercase variables used in the preceding examples are defined as follows:

msg_pds
: is the name of the install message file containing all the application's message skeletons for a specific language. *msg_pds* must be a partitioned data set.

msg_div_obj
: specifies the name of the run-time message file that is to contain the compiled message skeletons for the language. *msg_div_obj* must be a linear VSAM data set suitable for use as a data-in-virtual object.

lang,dbcs
: specifies two parameters. *lang* is the 3-character language code of the messages contained in the install message file. *dbcs* indicates whether this language contains double-byte characters. The values for *dbcs* are *y* for yes and *n* for no.

After creating the run-time message file by compiling the install message file, determine the amount of storage the run-time message file used. This calculation is necessary when compiling these messages in the system's run-time message file. The following JCL example shows you how to run a report showing the storage used.

```
//LISTCAT   JOB MSGLEVEL=(1,1)
//MCAT      EXEC PGM=IDCAMS,REGION=4096K
//SYSPRINT  DD SYSOUT=*
//SYSIN     DD *
  LISTCAT   LEVEL(msg_div_obj) ALL
/*
```

Checking the Message Compiler Return Codes

The message compiler generates a return code contained in register 15. The return codes are as follows:

Code	Meaning
0	Successful completion.
4	Processing complete. Run-time message file is complete but the compiler generated warnings.
8	Processing complete. The run-time message file is usable but incomplete.
12	Processing ended prematurely. The run-time message file is unusable.

You should correct all errors and recompile if you receive any return code other than 0.

Updating the System Run-Time Message File

After validating your application install message files, you can update the system run-time message file. You need to do the following:

- Add the application's install message files to the system's install message file, or add a DD statement to the JCL used to compile the system's install message files.
- Allocate a data set for the system run-time message file if it is not allocated yet. Use the storage requirements you received from running the IDCAMS report.
- Compile the system's install message files into the system run-time message files using the message compiler for each install message file. See "Compiling Message Files" on page 17-7.

You are using the same invocations to update the system run-time message file as you do to verify message skeletons. The difference is that the resulting system run-time message file is what MMS can use to translate messages for the system and applications.

Using MMS Translation Services in an Application

After you have compiled the translated messages and updated the system run-time message files, your program can use MMS services to retrieve translated message text. You need to do the following:

- Determine the language in which you want the application's messages translated, and use QRYLANG to check its availability.
- Use TRANMSG to retrieve translated messages.

You must also determine the action the application will take if the requested function does not complete, or if an output device cannot support the language.

Determining which Languages are Available (QRYLANG)

You need to determine if the language in which you want to issue messages is available to MMS. The message query function (QRYLANG) allows you to verify that the language you want is active, and also to receive a list of all available languages.

QRYLANG returns the information you request in the language query block (LQB), mapped by CNLMLQB. This block contains the following:

- The standard 3-character code for the language
- The name of the language
- A flag indicating whether the language contains double-byte characters

If you ask for a list of all available languages, QRYLANG returns an LQB with one language entry for each language.

You need to define storage for an LQB before issuing QRYLANG. To determine how much storage you need for the LQB if you want a list of all active languages:

- Calculate the length of the header section in mapping macro CNLLQB.
- Determine the total number of languages by looking in the MCAALCNT (active language count) field of the MCA, mapped by CNLMMCA.
- Multiply the total number of languages you intend to query by the LQBEBL (the length of one entry). This will give you the length of the LQB substitution data area.
- Add the length of the LQB substitution data area to the length of the header.

To determine how much storage you need for the LQB if you want to query one language:

- Calculate the length of the header section in mapping macro CNLLQB.
- Add the length of the LQB substitution data area to the length of the header.

Retrieving Translated Messages (TRANMSG)

In your application, call TRANMSG before the code that issues the message. TRANMSG obtains a message in the specified language and returns the message in a message text block (MTB). TRANMSG can obtain one message or several messages at a time. The application can then display the translated text. If the requested language is not available, TRANMSG returns the message unchanged. To check the availability of specific languages, use the QRYLANG macro described in "Determining which Languages are Available (QRYLANG)."

A message input/output block (MIO) serves as both input and output for TRANMSG. You can either build the MIO yourself or have TRANMSG do it for you. If you do not supply a formatted MIO, TRANMSG constructs one. See *Diagnosis: Data Areas* for information on the MIO. When you issue TRANMSG, the MIO must contain the following:

- The code of the language into which you want the message translated
- The addresses of the messages you want translated
- The address of an output buffer in the calling program's address space where TRANMSG is to return the translated message or messages

The application's messages can be in one of the following forms:

- Message text blocks (MTBs, mapped by CNLMMTB)
- Message parameter blocks (MPBs, mapped by CNLMMPB)
- Self-defined text (a 2-byte length field followed by message text)
- A combination of any of the three.

When TRANMSG completes, the MIO, mapped by CNLMMIO, contains the address of the translated message in the output buffer. The translated message is in the form of an MTB.

You can translate multi-line messages that contain continuation lines, that is, only the first line contains a message identifier in the skeleton. The following is an example of a multi-line message:

```
MSGID01 THIS IS LINE ONE OF THIS MULTI-LINE MESSAGE
        THIS IS LINE TWO OF THIS MULTI-LINE MESSAGE
        THIS IS LINE THREE OF THIS MULTI-LINE MESSAGE
```

Translating a multi-line message is a little different from translating a single-line message. You can take one of the following additional steps in preparing the message for translation:

- Add the message identifier to the beginning of the message text for each line subsequent to the first. For the example above, the modified message would appear as follows:

  ```
  MSGID01 THIS IS LINE ONE OF THIS MULTI-LINE MESSAGE
  MSGID01 THIS IS LINE TWO OF THIS MULTI-LINE MESSAGE
  MSGID01 THIS IS LINE THREE OF THIS MULTI-LINE MESSAGE
  ```

  When you invoke TRANMSG, MMS will process this message as three separate lines of text.

- Set the MIOCONT flag on the MIO message entry structure for lines subsequent to the first (lines two and three in the example). The MIOCONT flag tells MMS that a specific line of text is associated with the previous line of text. MMS associates the message identifier of the first line with the message text of the subsequent lines. *Assembler Programming Reference* provides a coding example that translates a multi-line message.

Using Message Parameter Blocks for New Messages (BLDMPB and UPDTMPB)

If you are writing a new application or adding new messages to an existing application, you can place the message text in the install message files rather than in the application code. To translate message text that exists only in the install message files, your program uses a message parameter block (MPB) containing a message identifier, format number, line number, and any substitution data. Using MPBs and install message files in this way provides the convenience of having to modify only the install message file if any of your message text requires a change. It also allows you to have a single repository for message text.

To build a message parameter block (MPB), allocate storage for the MPB, and issue BLDMPB and UPDTMPB. BLDMPB initializes the MPB and adds the fixed message data (called the message header), and UPDTMPB adds one substitution token to the MPB for each invocation.

Issue BLDMPB once for each MPB you will build and before you issue UPDTMPB. Issue UPDTMPB once for each substitution token in the message. You can also use UPDTMPB to replace or change the value of a particular substitution token in an existing MBP. However, you must ensure that the new value is not longer than the original value to maintain the integrity of the MPB. You might use UPDTMPB if you want to invoke TRANMSG several times with one MPB. For example, if you have an MPB associated with a message that you will translate in several languages, you can change only the language code in the MIO, and issue TRANMSG.

Once you have built an MPB for a message, you can issue TRANMSG to return the text of the message in a message text block (MTB). If the requested language is not available, TRANMSG returns the message number and its substitution data as a text string in the output area.

Support for Additional Languages

You can also translate messages into languages not currently available through IBM. You can do this in the following way:

- Select the language code in Figure 17-8 that matches the language into which you plan to translate messages. You will need that language code.
- If the messages you want to translate are MVS system messages, there may already be U.S. English skeletons for them, so all you need to supply are the translated skeletons. If the messages are from an application you have written, you need to supply both the English and translated skeletons. Follow the procedures described in "Creating Install Message Files" on page 17-2, "Validating Message Skeletons" on page 17-5, and "Updating the System Run-Time Message File" on page 17-9.
- Ask the installation's system programmer to:
  - Modify the parmlib member, MMSLSTxx, adding the language code.
  - Create a new config member, CNLcccxx, for the new language.
  - Restart MMS using the SET MMS command.

See *MVS Initialization and Tuning Reference* for more information on setting up config members, and parmlib members.

Figure 17-8 (Page 1 of 2). Languages available to MVS message service. These languages may not necessarily be available to your installation.

Code	Language Name	Principal Country
CHT	Traditional Chinese	R.O.C.
CHS	Simplified Chinese	P.R.C.
DAN	Danish	Denmark
DEU	German	Germany
DES	Swiss German	Switzerland
ELL	Greek	Greece
ENG	UK English	United Kingdom
ENU	US English	United States
ESP	Spanish	Spain

Figure 17-8 (Page 2 of 2). Languages available to MVS message service. These languages may not necessarily be available to your installation.

Code	Language Name	Principal Country
FIN	Finnish	Finland
FRA	French	France
FRB	Belgian French	Belgium
FRC	Canadian French	Canada
FRS	Swiss French	Switzerland
ISL	Icelandic	Iceland
ITA	Italian	Italy
ITS	Swiss Italian	Switzerland
JPN	Japanese	Japan
KOR	Korean	Korea
NLD	Dutch	Netherlands
NLB	Belgian Dutch	Belgium
NOR	Norwegian	Norway
PTG	Portuguese	Portugal
PTB	Brazil Portuguese	Brazil
RMS	Rhaeto-Romanic	Switzerland
RUS	Russian	USSR
SVE	Swedish	Sweden
THA	Thai	Thailand
TRK	Turkish	Turkey

For more information on translation, see *NLS Reference Manual*.

Example of an Application that Uses MMS Translation Services

The following example builds and updates and MPB, then invokes the MMS translate function to obtain the translated message. There are more examples for each MMS macro in the *Assembler Programming Reference*.

```
TRANSMPB CSECT
TRANSMPB AMODE  31
TRANSMPB RMODE  ANY
         STM    14,12,12(13)
         BALR   12,0
         USING  *,12
         ST     13,SAVE+4
         LA     15,SAVE
         ST     15,8(13)
         LR     13,15
*
*********************************************************************
*        OBTAIN WORKING STORAGE AREA                                *
*********************************************************************
         GETMAIN RU,LV=STORLEN,SP=SP230
```

```
         LR    R4,R1
*
*********************************************************************
*        CREATE MPB HEADER SECTION                                  *
*********************************************************************
*
         BLDMPB MPBPTR=(R4),MPBLEN=MPBL,MSGID=MSGID,                C
               MSGIDLEN=MIDLEN
*
*********************************************************************
*        ADD SUBSTITUTION DATA TO MPB FOR DAY 3, TUESDAY            *
*********************************************************************
*
         LR    R2,R4
         A     R2,MPBL
         USING VARS,R2
*
         UPDTMPB MPBPTR=(R4),MPBLEN=MPBL,SUBOOFST=VARS,             C
               TOKEN=TOKN,TOKLEN=TOKL,TOKTYPE=TOKT,                 C
               SUBSDATA=SDATA,SUBSLEN=SDATAL
*
         USING MIO,R3
         LA    R3,VARSLEN          OBTAIN LENGTH OF VARS AREA
         AR    R3,R2               CALCULATE ADDRESS MIO
         LA    R5,MLEN             GET LENGTH OF MIO
         AR    R5,R3               CALCULATE ADDRESS OF OUTPUT BUFFER
         ST    R4,VARSINBF         CREATE ADDRESS LIST
*
*********************************************************************
*        ISSUE TRANSLATE TO OBTAIN MESSAGE TEXT REPRESENTED BY THE  *
*        CREATED MPB                                                *
*********************************************************************
*
         TRANMSG MIO=MIO,MIOL=MIOLEN,INBUF=(VARSINBF,ONE),          C
               OUTBUF=(R5),OUTBUFL=OUTAREAL,LANGCODE=LC
*
*********************************************************************
*        FREE STORAGE AREA                                          *
*********************************************************************
*
         FREEMAIN RU,LV=STORLEN,SP=SP230,A=(4)
*
         L     13,SAVE+4
         LM    14,12,12(13)
         BR    14
         DROP
*********************************************************************
MPBL     DC    A(MPBLEN)
MSGID    DC    CL10'MSGID2'
MIDLEN   DC    A(MIDL)
TOKN     DC    CL3'DAY'
TOKL     DC    F'3'
TOKT     DC    CL1'3'
SDATA    DC    CL1'3'
SDATAL   DC    A(SDL)
MIOLEN   DC    A(MLEN)
OUTAREAL DC    A(STORLEN-(MPBLEN+VARSLEN+MLEN))
ONE      DC    F'1'
LC       DC    CL3'JPN'
SAVE     DC    18F'0'
```

```
SP230    EQU   230
STORLEN  EQU   512
SDL      EQU   6
MIDL     EQU   6
MPBLEN   EQU   (MPBVDAT-MPB)+(MPBMID-MPBMSG)+(MPBSUB-MPBSB)+MIDL+SDL
MLEN     EQU   (MIOVDAT-MIO)+MIOMSGL
R1       EQU   1
R2       EQU   2
R3       EQU   3
R4       EQU   4
R5       EQU   5
*********************************************************************
         DSECT
         CVT   DSECT=YES
         CNLMMPB
         IHASCVT
         CNLMMCA
         CNLMMIO
VARS     DSECT
VARSINBF DS    F
VARSAREA DS    CL24
VARSLEN  EQU   *-VARS
         END TRANSMPB
```

Chapter 18. Using Data Compression and Expansion Services

Data compression and expansion services allow you to compress certain types of data so that the data occupies less space while you are not using it. You can then restore the data to its original state when you need it.

Data that contains many repeat characters can exploit these services most effectively. Examples include:

- Data sets with fixed field lengths that might contain many blanks
- 80-byte source code images of assembler language programs.

Using these services with other types of data might not result in appreciable data volume reduction. In some cases, data volume might even be increased.

Services Provided

Data compression and expansion services, which your program invokes through the CSRCESRV macro, are described as follows:

- **Data Compression Service**

  This service compresses a block of data that you identify, and stores that data in compressed form in an output area. The service uses an algorithm called run length encoding[1] to compress the data. In some cases, the service uses an interim work area. Only programs running under MVS/ESA can use the data compression service.

- **Data Expansion Service**

  This service expands a block of data that you identify; the data must have been compressed by the data compression service. The data expansion service reverses the algorithm that the data compression service used, and stores the data in its original form in an output area. In some cases, the service uses an interim work area.

  The data expansion service is available in an MVS/ESA version and an MVS/XA version. Programs running under MVS/ESA can use either version. Programs running under MVS/XA can use only the MVS/XA version, and there are certain restrictions. See "Running under an MVS/XA System" on page 18-4 for further details.

- **Query Service**

  This service queries the system to determine the following:

  - Whether data compression is supported by the system currently installed
  - The size of the work area required by the compression or expansion service.

  To use the data compression and data expansion services, you need the information that the query service provides. Invoke the query service before invoking either the data compression or data expansion services.

[1] Run length encoding is a compression technique that compresses repeating characters, such as blanks.

The query service is available in an MVS/ESA version and an MVS/XA version. Programs running under MVS/ESA can use either version. Programs running under MVS/XA can use only the MVS/XA version, and there are certain restrictions. See "Running under an MVS/XA System" on page 18-4 for further details.

Figure 18-1 summarizes the services available, and the systems under which these services can run.

SYSTEM	MVS/ESA Version of Service	MVS/XA Version of Service
MVS/ESA	Compression Expansion Query	Expansion Query
MVS/XA	Not applicable	Expansion Query

Figure 18-1. Summary of Data Compression and Expansion Services

Note: These services do not provide a recovery routine (for example, ESTAE or FRR) because it would not contribute to reliability or availability.

Running under an MVS/ESA System

Programs running under MVS/ESA can access the MVS/ESA version of the data compression, data expansion, and query services. Programs running under MVS/ESA can also access the MVS/XA version of the data expansion and query services.

Using the MVS/ESA Version of the Services

The MVS/ESA version of the data compression, data expansion, and query services resides in SYS1.LPALIB. Your program can invoke these services by using the CSRCESRV macro.

See *Assembler Programming Reference* for complete instructions on how to use the CSRCESRV macro.

To invoke the data compression or data expansion services, follow these steps:

1. Invoke the query service by coding the CSRCESRV macro specifying SERVICE=QUERY. The macro returns the information you need to invoke the data compression or data expansion service.
2. If you plan to compress data, check the information returned to ensure that compression is supported.
3. Invoke the data compression service (or the data expansion service) by coding the CSRCESRV macro specifying SERVICE=COMPRESS (or SERVICE=EXPAND).

Using the MVS/XA Version of the Services

Programs running under MVS/ESA can invoke the MVS/XA version of the query service and the data expansion service. You might use the MVS/XA version if you write a program that must expand data on both MVS/ESA and MVS/XA. You can invoke the MVS/XA version in the same manner when running under MVS/ESA as you do when running under MVS/XA. See "Running under an MVS/XA System" on page 18-4 for details on invoking the MVS/XA version.

An alternative approach for a program that must expand data on both MVS/ESA and MVS/XA is to test the level of the MVS system at execution time to determine which version of the services to use. Figure 18-2 provides an example of the code you might use to conduct this test.

```
             ⋮
*
* TEST FOR MVS/ESA
*
         TM    CVTDCB,CVTOSEXT
         BNO   SP2CHK
         TM    CVTOSLV0,CVTXAX
         BNO   SP2CHK
              ⋮
*
* TEST FOR MVS/XA
*
SP2CHK   TM    CVTDCB,CVTMVSE
         BNO   SP1
              ⋮
*
* CONTINUE CODING
*
SP1      ...
              ⋮
*
* MAP THE COMMUNICATIONS VECTOR TABLE (CVT)
*
         CVT   DSECT=YES
              ⋮
*
```

Figure 18-2. Testing the Level of the MVS System at Execution Time

Running under an MVS/XA System

Programs running under MVS/XA can use the MVS/XA version of the query service and the data expansion service, but there is no MVS/XA version of the data compression service. Therefore, to expand data under MVS/XA, the data must have been compressed by a program running under MVS/ESA. Programs running under MVS/XA must also adhere to the following restrictions:

- To assemble, include the correct level (Version 3 Release 1.3 or later) of SYS1.MACLIB in the SYSLIB concatenation for the assembly step.
- To execute, include the correct level (Version 3 Release 1.3 or later) of SYS1.MIGLIB in the STEPLIB (or JOBLIB) of the JCL to execute the program. (See *JCL User's Guide* for further information about STEPLIB and JOBLIB DD statements.)

Programs running under MVS/XA may use the CSRCESRV macro, but the macro will not automatically locate the entry point address for the service requested. Your program must supply that information. To expand data on an MVS/XA system, here are the steps you must follow. (See *Assembler Programming Reference* for complete instructions on how to use the CSRCESRV macro.)

1. Load the CSRCEXA load module from SYS1.MIGLIB, using the LOAD macro. You must do so while your program is in task mode. See *Assembler Programming Reference* for LOAD macro syntax.
2. Save the entry point address and place it in a general purpose register (GPR) so you can pass it to the CSRCESRV macro.
3. Invoke the query service by coding the CSRCESRV macro specifying SERVICE=QUERY and VECTOR=(*reg*), where *reg* is the GPR containing the entry point address of CSRCEXA. The macro returns the information you need to invoke the data expansion service.
4. Invoke the data expansion service by coding the CSRCESRV macro specifying SERVICE=EXPAND and VECTOR=(*reg*), where *reg* is the GPR containing the entry point address of CSRCEXA.

Chapter 19. Accessing Unit Control Blocks (UCBs)

Each device in a configuration is represented by a unit control block (UCB). In a dynamic configuration environment, a service obtaining UCB information needs to be able to detect any changes in the configuration that could affect the returned information. The MVS I/O configuration token provides this capability. You can scan UCBs with the UCBSCAN macro to obtain information about the devices in the configuration.

The eligible device table (EDT) contains the definitions for the installation's device groups. The EDTINFO macro allows you to obtain information from the EDT.

Detecting I/O Configuration Changes

You can use the MVS I/O configuration token to detect I/O configuration changes. The MVS I/O configuration token is a 48-byte token that uniquely identifies an I/O configuration to the system. The token will change whenever the software configuration definition changes. Thus, if your program obtains the current MVS I/O configuration token and compares it to one previously obtained, the program can determine whether there has been a change in the I/O configuration: If the tokens do not match, the I/O configuration has changed.

An optional parameter, IOCTOKEN, is available with the UCBSCAN macro. Specifying IOCTOKEN ensures that the system will notify the caller through a return code and will not return any data if the current I/O configuration is not consistent with the configuration represented by the token specified as input by the caller.

Use the following ways to obtain the current MVS I/O configuration token:

- Issue the IOCINFO macro.
- Issue the UCBSCAN macro, setting the input specified by the IOCTOKEN parameter to binary zeroes. The macro will then return the current I/O configuration token at the start of the scan.
- Issue EDTINFO macro, setting the input specified by the IOCTOKEN parameter to binary zeroes.

Use of the MVS I/O configuration token can help prevent data inconsistencies that might occur if the I/O configuration changes between the time the caller obtained the token and the time the service returns the information. For example, you can use the configuration token to determine whether the I/O configuration changes during a UCB scan. If the IOCTOKEN parameter is specified with UCBSCAN, the caller will be notified through a return code if the set of UCBs changes while the scan is in progress. Checking for this return code allows the caller to restart the scan to ensure that copies of all UCBs in the current configuration are obtained.

An unauthorized program can use the MVS I/O configuration token to regularly check whether a configuration change has occurred, as in the following example:

- The program issues the IOCINFO macro to obtain the MVS I/O configuration token.
- The program sets a time interval that is to expire in 10 minutes, using the STIMER macro.

- When the time interval expires, the user-specified timer exit routine gets control and issues the IOCINFO macro to obtain the MVS I/O configuration token that is current at this later time.
- The program compares the newly-obtained token with the original one.
- If the tokens match, no I/O configuration change has occurred, and the program resets the time interval for another 10 minutes to check again at that time.
- If the tokens do not match, a configuration change has occurred. The program then rebuilds its control structures by using the UCBSCAN macro, specifying the IOCTOKEN parameter to check for any further I/O configuration changes while the rebuilding process is in progress. After the control structures are rebuilt for the new I/O configuration, the program resets the time interval for 10 minutes to check again for I/O configuration changes

Scanning UCBs

You can use the UCBSCAN macro to scan UCBs. On each invocation, UCBSCAN returns a copy of a UCB in a user-supplied work area. The scan can include all UCBs in the system, or be restricted to a specific device class. For example, you could use UCBSCAN to find all DASD devices currently defined to the configuration. It is also possible to restrict the scan to UCBs for static and installation-static devices, or to include UCBs for dynamic devices as well.

Example of a Program That Obtains Copies of All the UCBs: This example program obtains copies of all UCBs, including those for devices defined as dynamic. It uses the MVS I/O configuration token to determine if the I/O configuration changes during the scan, and it restarts the scan if the I/O configuration has changed. On each invocation of UCBSCAN, the system returns a copy of a UCB at the address specified by UCBAREA and return the current MVS I/O configuration token.

```
SCANEXMP CSECT
SCANEXMP AMODE 31
SCANEXMP RMODE ANY
         DS    0H
         BAKR  R14,0                        Save regs on linkage stack
         LR    R12,R15                      Set up code reg
         USING SCANEXMP,R12
         LA    R13,SAVEAREA                 Get save area address
         MVC   SAVEAREA+4(4),FIRSTSAV       First save area in chain
*
*        ...
*
RESCANLP DS    0H
         IOCINFO IOCTOKEN=TOKEN             Get current IOCTOKEN
         XC    SCANWORK,SCANWORK            Clear scan work area
SCANLOOP DS    0H
         UCBSCAN UCBAREA=UCBCOPY,WORKAREA=SCANWORK,DYNAMIC=YES,        +
               IOCTOKEN=TOKEN
         LTR   R15,R15                      Was a UCB returned?
         BNZ   SCANDONE                     No, either a configuration
*                                           change has occurred
*                                           or no more UCBs
*
*    Process  UCB
*
         B     SCANLOOP
SCANDONE DS    0H
         LA    R02,12                       Return code for a
*                                           configuration change
         CR    R15,R02                      Did a configuration change
*                                           occur?
         BE    RESCANLP                     Yes, start scan again
FINISHED DS    0H
*
*        ...
*
ENDIT    DS    0H
         PR                                 Return to caller
         EJECT
*
*   Register equates
*
R02      EQU   2
R03      EQU   3
R09      EQU   9
R12      EQU   12
R13      EQU   13
R14      EQU   14
R15      EQU   15
         DS    0F
FIRSTSAV DC    CL4'F1SA'                    First save area ID
SAVEAREA DS    18F                          Save area
TOKEN    DS    48C                          IOCTOKEN area
UCBCOPY  DS    48C                          UCB Copy returned by
*                                           SCAN
SCANWORK DS    CL100                        Work area for SCAN
         END   SCANEXMP
```

Obtaining Eligible Device Table Information

The installation's device groups are defined in the eligible device table (EDT). An EDT is an installation-defined and named representation of the devices that are eligible for allocation. This table also defines the relationship of generic device types and esoteric group names. The term "generic device type" refers to the general identifier IBM gives a device; for example, 3380. An esoteric device group is an installation-defined and named collection of I/O devices; TAPE is an example of an esoteric group name. See *System Modifications* for further information on the EDT and allocation considerations.

The EDTINFO macro enables you to obtain information from the EDT and to check your device specification against the information in the EDT. You can use the EDTINFO macro to perform the following functions:

- Check groups. The EDTINFO macro determines whether the input device numbers constitute a valid allocation group. The device numbers are a valid allocation group if either of the following is true:
  - For any allocation group in the EDT that contains at least one of the device numbers specified in the input device number list, **all** of the device numbers in that group in the EDT are contained in the input device number list
  - None of the allocation groups in the EDT contain any of the numbers specified in the input device number list.

  If neither of these is the case, the device numbers are not a valid allocation group.
- Check units. The EDTINFO macro determines whether the input device numbers correspond to the specified unit name. The unit name is the EBCDIC representation of the IBM generic device type or esoteric group name.
- Return unit name. The EDTINFO macro returns the unit name associated with the UCB device type provided as input.
- Return unit control block (UCB) addresses. The EDTINFO macro returns a list of UCB addresses associated with the unit name or device type provided as input.

  Note: The EDTINFO macro returns UCB addresses only for static and installation-static UCBs.
- Return group ID. The EDTINFO macro returns the allocation group ID corresponding to each UCB address specified as input.
- Return attributes. The EDTINFO macro returns general information about the unit name or device type specified as input.
- Return unit names for a device class. The EDTINFO macro returns a list of generic device types or esoteric group names associated with the device class specified as input.
- Return UCB device number list. The EDTINFO macro returns the UCB device number list associated with the unit name or UCB device type specified as input. You can also specify that devices defined to the system as dynamic are to be included in the list.

- Return maximum eligible device type. The EDTINFO macro returns the maximum eligible device type (for the allocation and cataloging of a data set) associated with the unit name or device type, recording mode, and density provided as input. The maximum eligible device type is the tape device type that contains the greatest number of eligible devices compatible with the specified recording mode and density.

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Numerics

Special Characters

Reader's Comments

MVS/ESA
Application Development Guide:
Assembler Language Programs

MVS/ESA System Product:
JES2 Version 4
JES3 Version 4

Publication No. GC28-1644-1

Use this form to tell us what you think about this manual. If you have found errors in it, or if you want to express your opinion about it (such as organization, subject matter, appearance) or make suggestions for improvement, this is the form to use.

To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer. This form is provided for comments about the information in this manual and the way it is presented.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Be sure to print your name and address below if you would like a reply.

Name

Address

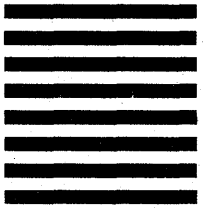
Company or Organization

Phone No.

Cut
Alor

Fold and Tape **Please do not staple** Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department D58, Building 921-2
PO BOX 950
POUGHKEEPSIE NY 12602-9935

Fold and Tape **Please do not staple** Fold and Tape

Cut
Alor

Reader's Comments

MVS/ESA
Application Development Guide:
Assembler Language Programs

MVS/ESA System Product:
JES2 Version 4
JES3 Version 4

Publication No. GC28-1644-1

Use this form to tell us what you think about this manual. If you have found errors in it, or if you want to express your opinion about it (such as organization, subject matter, appearance) or make suggestions for improvement, this is the form to use.

To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer. This form is provided for comments about the information in this manual and the way it is presented.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Be sure to print your name and address below if you would like a reply.

Name

Address

Company or Organization

Phone No.

Cut
Alo

Fold and Tape **Please do not staple** Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department D58, Building 921-2
PO BOX 950
POUGHKEEPSIE NY 12602-9935

Fold and Tape **Please do not staple** Fold and Tape

Cu
Alc

Reader's Comments

MVS/ESA
Application Development Guide:
Assembler Language Programs

MVS/ESA System Product:
JES2 Version 4
JES3 Version 4

Publication No. GC28-1644-1

Use this form to tell us what you think about this manual. If you have found errors in it, or if you want to express your opinion about it (such as organization, subject matter, appearance) or make suggestions for improvement, this is the form to use.

To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer. This form is provided for comments about the information in this manual and the way it is presented.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Be sure to print your name and address below if you would like a reply.

Name

Address

Company or Organization

Phone No.

IBM®

Cut
Alor

Fold and Tape | **Please do not staple** | Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department D58, Building 921-2
PO BOX 950
POUGHKEEPSIE NY 12602-9935

Fold and Tape | **Please do not staple** | Fold and Tape

Cut
Alo

Reader's Comments

MVS/ESA
Application Development Guide:
Assembler Language Programs

MVS/ESA System Product:
JES2 Version 4
JES3 Version 4

Publication No. GC28-1644-1

Use this form to tell us what you think about this manual. If you have found errors in it, or if you want to express your opinion about it (such as organization, subject matter, appearance) or make suggestions for improvement, this is the form to use.

To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer. This form is provided for comments about the information in this manual and the way it is presented.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Be sure to print your name and address below if you would like a reply.

Name

Address

Company or Organization

Phone No.

Cu
Al

Fold and Tape | **Please do not staple** | Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

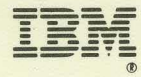
FIRST CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department D58, Building 921-2
PO BOX 950
POUGHKEEPSIE NY 12602-9935

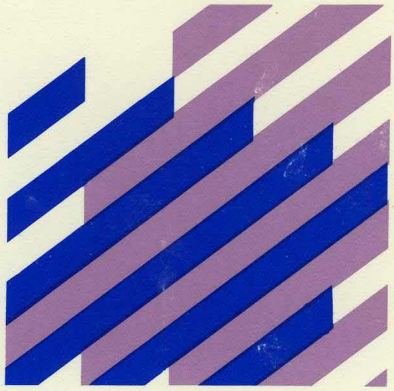
Fold and Tape | **Please do not staple** | Fold and Tape

Cu
Al



File Number:S370 / S390-36
Program Numbers: 5695-047
5695-048

Printed in U.S.A.



GC28-1644-01